Titre: Title:	Méthodologie orientée performance applicable à la validation d'algorithmes de traitement vidéo et de leur implémentation matérielle
Auteur: Author:	Serge Catudal
Date:	2005
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Catudal, S. (2005). Méthodologie orientée performance applicable à la validation d'algorithmes de traitement vidéo et de leur implémentation matérielle [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7597/

UNIVERSITÉ DE MONTRÉAL

MÉTHODOLOGIE ORIENTÉE PERFORMANCE APPLICABLE À LA VALIDATION D'ALGORITHMES DE TRAITEMENT VIDÉO ET DE LEUR IMPLÉMENTATION MATÉRIELLE

SERGE CATUDAL
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(MICROÉLECTRONIQUE)
JUILLET 2005

Your file Votre référence
ISBN: 978-0-494-16763-2
Our file Notre référence
ISBN: 978-0-494-16763-2

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

MÉTHODOLOGIE ORIENTÉE PERFORMANCE APPLICABLE À LA VALIDATION D'ALGORITHMES DE TRAITEMENT VIDÉO ET DE LEUR IMPLÉMENTATION MATÉRIELLE

présenté par: CATUDAL Serge

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. NICOLESCU Gabriela, Ph.D., président

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. BOIS Guy, Ph.D., membre et codirecteur de recherche

M. KHOUAS Abdelhakim, Ph.D., membre

"Il est rare de trouver un homme qui se livre trois ans à l'étude, sans avoir en vue un salaire."

- Confucius

REMERCIEMENTS

Je tiens à remercier Monsieur Yvon Savaria, mon directeur de maîtrise, pour sa disponibilité, ses conseils et son analyse qui m'ont permis de mener à terme mes travaux de recherche sous les meilleurs conditions. Je tiens de plus à remercier mon co-directeur Guy Bois. Je tiens à souligner le support financier de Micronet ainsi que de Gennum Corporation et je tiens à remercier Jean-Marc Tremblay et Arnold Veenstra de Gennum Corporation pour leurs conseils techniques apportés lors des conférences téléphoniques tout au long de ma recherche.

J'en profite pour remercier tous mes camarades du GRM (Groupe de Recherche en Microélectronique) et en particulier Bruno Tanguay, Louis-Pierre Lafrance, Tien Hung Bui, Alexandre Chureau, Mame Maria Mbaye, Ghyslain Provost, Robert Grou ainsi que tous mes camarades du CIRCUS, soit Francis St-Pierre, Françoit Deslauriers, Simon Provost et Jean-François Thibeault. Je tiens aussi à remercier Sébastien Regimbal pour son aide dans le domaine de la vérification de design matériel, sans oublier les techniciens du GRM Alexandre Vesey et Régeant Lepage et notre secréraire Ghyslaine Éthier Carrier.

Je terminerai par remercier ma soeur Marcelle Catudal pour son support moral et financier dans les moments opportuns. Je tiens de plus à remercier mon cousin Francis et mon ami Oktay Ali Yildiz et sa fiancée Amélie pour leur support moral tout au long de mes études.

RÉSUMÉ

Les architectures de traitement vidéo sont généralement dérivées à partir d'algorithmes validés auparavant à l'aide de langage de haut niveau comme le C/C++ ou *Matlab/Simulink*. Elles sont généralement simulées en résolution virgule-flottante qui est une résolution très coûteuse à utiliser pour la mise en oeuvre de matériels dédiés. Une bonne façon de minimiser ou d'optimiser la complexité d'une implémentation matérielle en ce qui a trait à sa performance de calcul et à sa consommation de puissance, est d'utiliser une précision en virgule-fixe. De plus, certains algorithmes de traitement vidéo contiennent des paramètres devant être fixés afin d'obtenir une performance optimale. Ces paramètres sont généralement déterminés empiriquement.

La traduction d'une formulation en virgule-flottante vers une formulation en virgule-fixe, pour une implémentation matérielle, peut causer certains problèmes. Par exemple, la précision de chaque opérande et la valeur de chaque paramètre contenu dans l'algorithme devraient être sélectionnées minitieusement afin de préserver la stabilité de l'algorithme ainsi que l'acuité de son calcul. De plus, cette sélection peut détériorer le comportement d'un algorithme de traitement vidéo en produisant des artéfacts visuels déplaisant pour un observateur humain.

Une nouvelle méthode systématique et automatique fut développée afin de valider des algorithmes de traitement vidéo. Le but de cette méthode est de pousser la tâche de la validation d'algorithme de traitement vidéo à un niveau d'abstraction plus élevé pour un concepteur matériel. Afin d'atteindre ce but, la nouvelle méthode proposée utilise une plateforme de validation automatique d'algorithmes de traitement vidéo prenant avantage d'une nouvelle métrique mesurant la qualité d'image. L'utilisation de métriques objectives mesurant la qualité d'image permet

de retirer un observateur humain du processus d'évaluation de la qualité d'image. Les résultats expérimentaux obtenus à l'aide de trois filtres spatiaux ainsi que deux algorithmes de dé-entrelacement vidéo sont présentés afin d'illustrer la validité de cette approche.

ABSTRACT

Video Processing architectures are usually derived from algorithms previously validated with C/C++ or *Matlab/Simulink* implementations. They are usually simulated in floating-point resolution, which is very expensive in terms of hardware cost. A good way to minimize or optimize hardware implementation complexity, computational performance, and power consumption is to use fixed-point precision. Furthermore, some video processing algorithms contain parameters that need to be set for optimal performance. These parameters are usually determined empirically.

The translation from a floating-point to a fixed-point formulation, for a hardware implementation, may cause some problems. For instance, the precision of each operand and the value of each parameter contained in the algorithm should be carefully selected to preserve algorithm stability and processing accuracy. Furthermore, this selection can deteriorate the behavior of a video processing algorithm to produce visual effects annoying to human observers.

A new systematic and automatic method was developed to validate video processing algorithm. The aim of this method is to push the task of video processing algorithm validation at a higher level of abstraction for a hardware designer. In order to achieve this goal, the new method proposed uses an automatic video processing validation platform that takes advantage of new objective image quality metrics. The use of objective image quality metrics allows removing human observers from the process of evaluating image quality. Experimental results obtained with three spatial filters and two de-interlacing algorithms are presented to illustrate the validity of this approach.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES NOTATIONS ET DES SYMBOLES

AMPDF:	Adapdive Minimum Pixel Difference Filter
ASIC:	Application Specific Integrated Circuit
AWLDT:	Automatic Word Length Determination Tool
CDFG:	Control Data Flow Graph
CTL:	Computational Tree Logic
DCT:	Discrete Consine Transform
DSP:	Digital Signal Processing
DVD:	Digital Versatile Disc
ELA:	Edge-based Line Average
FR:	Full Refernce metric
HDTV:	High Definition Television
HVS:	Human Visual System
HW:	Hardware
IEEE:	Institute of Electrical and Electronics Engineers
IRATIO:	Inverse Ratio
JPEG:	Joint Photographic Experts Group
LTL:	Linear Temporal Logic
MMSE:	Minimum Mean Square Error
MSSIM:	Mean Structural SIMilarity
MPEG:	Moving Picture Experts Group
NR:	No Reference metric
PT:	Pixel Threshold
OIQM:	Objective Image Quality Metric
RR:	Reduced Reference metric
RTL:	Register Transfer Level

SDTV:	Standard Definition Television
SoC:	System-on-a-Chip
SUSAN:	Smallest Univalue Segment Assimilating Nucleus
VHDL:	Very high speed integrated circuit Hardware Description Language

LISTE DES TABLEAUX

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

La conception de modules de traitement vidéo peut être divisée en deux composantes complémentaires associées au contrôle et au flot de données respectivement. Le but principal du contrôle est d'alimenter le chemin de données avec les bonnes données au bon moment, ainsi que de collecter les résultats sortant de celui-ci. Le but principal du chemin de données est d'effectuer la transformation de données spécifiée par l'algorithme de traitement vidéo. Les architectures matérielles d'algorithmes de traitement vidéo implémentant la transformation de données sont généralement dérivées à partir d'algorithmes préalablement validés à l'aide d'implémentations C/C++ ou *Matlab/Simulink*. Ces algorithmes sont généralement simulés à l'aide d'une résolution en virgule-flottante. Comme les implémentations en résolution virgule-flottante sont relativement très coûteuses en terme de coût matériel, une excellente méthode afin de minimiser ou d'optimiser la complexité d'une implémentation matérielle, sa performance ainsi que sa consommation de puissance, est d'utiliser une précision en virgule-fixe.

1.1 Éléments de la problématique

La traduction d'une formulation en virgule-flottante vers une formulation en virgule-fixe, une étape du processus qui conduit à une implémentation matérielle, peut causer quelques problèmes. Par exemple, la précision de chaque opérande devrait être attentivement sélectionnée afin de préserver la stabilité de l'algorithme ainsi que l'exactitude de son traitement. La sensibilité de la taille des mots sur

l'algorithme visé n'est pas claire *a priori*. De plus, cette traduction peut détériorer le comportement d'un algorithme de traitement vidéo en produisant des artéfacts visuels déplaisant pour un observateur humain. Une méthode systématique afin de valider l'implémentation d'un algorithme de traitement vidéo devient alors très intéressante.

La validation aide un concepteur de matériel à concevoir et à implémenter des systèmes complexes en assurant, avec un haut niveau de confiance, que ces derniers sont conformes à leur spécification dans toutes les circonstances. Il existe des techniques formelles afin de valider des sytèmes, mais celles-ci sont applicables seulement pour la validation du contrôle d'un système, i.e. ne sont pas applicables pour la validation du chemin de données d'un système. De plus, la majorité des travaux concernant la validation d'algorithmes de traitement vidéo portent sur des techniques de vérification formelle de transformation de boucle à l'intérieur d'un algorithme [14] et sur la validation au niveau système de la spécification d'algorithmes de traitement d'images dans leur environnement cible à l'aide d'émulation [20]. Cependant, ces techniques de validation ne sont pas utiles pour confirmer automatiquement l'absence d'artéfacts visuels déplaisant pour un observateur humain à l'intérieur d'une implémentation pratiquement correcte.

Par la suite, un grand nombre d'algorithmes de traitement vidéo sont contrôlés par des paramètres devant être fixés afin de pouvoir les utiliser. Un bon ajustement de ces paramètres peut avoir, encore une fois, un impact significatif sur la performance et le comportement de l'algorithme. Ces paramètres sont habituellement déterminés empiriquement par les concepteurs de matériel [22] [29] [37]. Il y a donc ici un besoin pour une méthodologie afin d'automatiser la détermination et la validation des paramètres contrôlant les algorithmes de traitement vidéo.

1.2 Objectifs de recherche

Le but de ce mémoire est de développer une méthode automatique et systématique pour la validation d'algorithmes de traitement vidéo. Cette méthode doit pousser à un niveau d'abstraction plus élevé le processus de validation d'algorithme de traitement vidéo pour un concepteur de matériel. De plus, cette méthode doit être robuste, et doit idéalement évaluer automatiquement la performance des algorithmes de traitement vidéo, et toujours produire des images valides, sans avoir recours à un observateur humain pour la tâche de l'évaluation des images résultantes.

Par la suite, la méthodologie développée devra être étendue afin d'estimer automatiquement l'ensemble des paramètres contrôlant un algorithme de traitement vidéo. Bref, cette extension doit permettre, dans un premier temps, d'évaluer les valeurs optimales des paramètres de l'algorithme lorsque toutes ses opérandes sont en résolution virgule-flottante. Dans un deuxième temps, cette extension doit automatiquement optimiser et valider l'implémentation matérielle de l'algorithme en résolution virgule-fixe et ajuster à la valeur optimale les paramètres de contrôle en tenant compte de la précision finie du chemin de données.

1.3 Organisation du mémoire

Le chapitre 2 de ce mémoire présente une revue de littérature sur les différents aspects concernant la méthodologie automatique et systématique afin de valider des algorithmes de traitement vidéo. Ce chapitre passe en revue les métriques objectives pour la mesure de la qualité d'image. Par la suite, la différence entre la validation formelle et la vérification fonctionnelle est présentée. Il se termine par la présentation des algorithmes de traitement vidéo servant à valider la méthodologie présentée au prochain chapitre.

Ensuite, le chapitre 3 présente la plateforme de validation d'algorithmes de traitement vidéo et ses différentes composantes. La composante clé de cette plateforme repose sur la combinaison d'un outil de détermination automatique de la taille de mot d'une opérande (AWLDT) [7] [8] avec une nouvelle métrique pouvant rejeter des solutions générant des images distortionnées. Cette nouvelle métrique est présentée dans ce chapitre, ainsi que les modifications apportées à l'outil AWLDT afin d'estimer automatiquement les paramètres d'un algorithme de traitement vidéo.

Le chapitre 4 présente deux techniques afin d'accélérer la validation d'algorithme de traitement vidéo. Afin d'effectuer cette étude, ce chapitre utilise comme modèle de référence un algorithme de dé-entrelacement adaptatif au mouvement. Ce dernier est composé de trois algorithmes distincts fonctionnant en parallèle.

Le chapitre 5 présente la conclusion de ce mémoire. Un résumé des travaux accomplis est présenté, suivi par la suite d'indications sur des travaux de recherche futurs pertinent sur le sujet de ce mémoire.

CHAPITRE 2

CONCEPTS DE BASE

Le but de ce travail est de développer une méthodologie automatique et systématique afin de valider des algorithmes de traitement vidéo et leur implémentation en matériel. Une revue des concepts de base sur les différents aspects de cette méthodologie devient donc nécessaire afin de bien comprendre les concepts sur lesquels celle-ci est basée. Ce chapitre est divisé en quatre sections principales. La section 2.1 présente les métriques objectives pour la mesure de la qualité d'image, alors que la section 2.2 présente la différence entre la validation et la vérification fonctionnelle. Ensuite, la section 2.3 présente les algorithmes de traitement vidéo utilisés à l'intérieur de ce travail et la section 2.4 conclut le chapitre.

2.1 Métriques objectives pour la mesure de la qualité d'image

Le but recherché avec la mesure objective de qualité d'images et de flots vidéo vise à élaborer des métriques pouvant estimer de façon quantitative et automatique leur qualité telle que perçue par un observateur humain [45]. En général, il existe trois situations possibles pour lesquelles une métrique de mesure de qualité d'images serait employée. La première situation consiste à surveiller la qualité des images pour des sytèmes de contrôle de qualité. Ensuite, la deuxième situation consiste à utiliser une telle métrique comme critère de référence pour des algorithmes de traitement d'images. La troisième et dernière situation consiste à incorporer une métrique mesurant la qualité d'images à l'intérieur d'un système de traitement

d'image qui optimise automatiquement des algorithmes et leurs paramètres.

Par la suite, les métriques objectives mesurant la qualité d'image (OIQM, objective image quality metric) peuvent être classifiées en trois catégories [9] [42] [45]. La première catégorie est nommée référence complète (FR, full reference) [44]. Les OIQM faisant partie de la catégorie FR, soit la majorité des métriques se trouvant dans la littérature, utilisent une image de référence, considérée comme étant non déformée, afin de la comparer avec l'image distortionnée. Donc, dans le cas des métriques FR, comme toute l'information de l'image originale est disponible, la mesure de la qualité retournée est alors plus précise et robuste [9]. La deuxième catégorie est nommée sans référence (NR, no reference). Cette catégorie est très pratique dans le cas où l'image de référence n'est pas disponible. En effet, les métriques NR extraient des caractéristiques de l'image afin de procéder à la mesure de la qualité de l'image distortionnée. Cependant, les métriques de cette catégorie sont présentement efficaces seulement dans le cas où le type de bruit à l'intérieur des images distortionnées est connu *a priori* [39]. La troisième et dernière catégorie est nommée référence réduite (RR) [46] et elle provient d'une solution se situant entre les modèles FR et NR. Ces métriques sont conçues afin de prédire la qualité perçue d'une image distortionnée à partir uniquement d'informations partielles de l'image de référence.

La motivation de l'utilisation de métriques objectives mesurant la qualité d'images à l'intérieur de ce mémoire provient du fait que la plateforme de validation d'algorithmes de traitement vidéo, présentée au Chapitre 3, utilise les mesures de qualité d'image comme critère de référence pour des algorithmes de traitement vidéo. Comme le but premier de cette plateforme est de retirer un observateur humain du processus de mesure de la qualité d'image, celle-ci a donc besoin d'une métrique de qualité objective afin de faire une différence cohérente entre une image originale et une image distortionnée. De plus, comme l'image de référence

est disponible à l'intérieur de la plateforme de validation d'algorithmes de traitement vidéo et que celle-ci doit être générique, seules les métriques de la catégorie FR ont été considérées. Celles-ci peuvent être divisées en deux groupes. La section 2.1.1 présente le premier groupe, soit celui mesurant les erreurs entre une image de référence et une image distortionnée. Par la suite, la section 2.1.2 présente le deuxième groupe, soit celui mesurant les similarités entre une image de référence et une image distortionnée.

2.1.1 Mesure de l'erreur dans l'image

Une image ou un signal vidéo, dont la qualité est évaluée, peut être représentée comme la somme d'un signal de référence et d'un signal d'erreur. Dans ce cas, la perte de qualité à l'intérieur de l'image proviendrait directement de l'intensité du signal d'erreur. Une méthode naturelle afin de mesurer la qualité à l'intérieur d'une image est de quantifier l'erreur entre un signal distortionné et son signal de référence. Les OIQM de la catégorie FR les plus largement utilisées et implémentant ce concept sont la métrique Mean Square Error (MSE) et la métrique Peak Signal-to-Noise Ratio (PSNR). Ces métriques sont décrites par les équations (2.1) et (2.2) respectivement.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2, \tag{2.1}$$

$$PSNR = 10\log_{10}\frac{L^2}{MSE}. \tag{2.2}$$

où x_i et y_i représentent les pixels de l'image source et de l'image distortionnée respectivement, N représente le nombre total de pixels dans l'image et L représente l'intensité maximale que peut prendre un pixel, soit la valeur 255 dans le cas de l'utilisation d'une résolution à 8-bits par pixel.

Cependant, ces métriques ne corrèlent pas très bien avec la perception de la qualité d'image d'un observateur humain [38] [45]. Il a été démontré dans [45] que différents types de bruits appliqués sur une image, perçus par un observateur humain comme ayant des niveaux de sévérité variables, pouvaient donner le même valeur de métrique MSE. C'est pourquoi plusieurs métriques mesurant la sensibilité aux erreurs à l'intérieur d'une image furent développées en se basant sur des modèles mathématiques du Système Visuel Humain (HVS).

En général, ces métriques utilisent une structure algorithmique, illustrée à la figure 2.1, basée sur cinq étapes. Lors de l'étape un, des opérations de pré-traitement telles que la transformation de l'espace de couleur, la calibration des appareils de visualisation, etc, sont effectuées. Par la suite, un filtrage des fonctions de sensibilité de contraste est efectué lors de l'étape deux. L'étape trois de cette structure sert à séparer les stimulis visuels en sous-domaines de fréquence à partir de différents modèles de décomposition de fréquence, e.g. Daly [15], Lubin [30], Wavelet [46], Tranformée en Cosinus Discrète (DCT) [1], etc. Suite à cette décomposition, l'étape quatre normalise chaque canal et l'étape cinq de cette structure combine les signaux d'erreur, provenant des diférents canaux, en un seul signal.

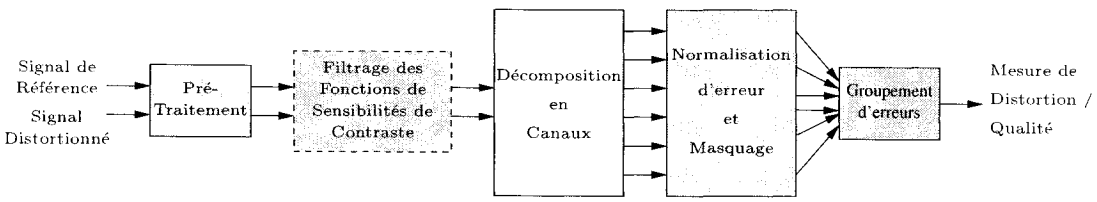


Figure 2.1 Diagramme de la structure générale des métriques basées sur la mesure d'erreur.

2.1.2 Mesure des similarités à l'intérieur d'une image

Le deuxième groupe base sa philosophie sur l'hypothèse que le système visuel humain est hautement adapté à l'extraction d'informations structurelles à partir de

son champ de vision [43]. Donc, la mesure du changement d'information structurelle à l'intérieur d'une image pourrait fournir une bonne approximation de la distortion perçue de celle-ci. C'est la raison pour laquelle ce type de métriques est basée sur la mesure de la perte d'information structurelle perçue à l'intérieur d'une image distortionnée, au lieu d'estimer l'erreur perçue afin de représenter la dégradation de celle-ci. La section 2.1.2.1 présente une métrique de ce type que nous avons utilisé dans le cadre de ce travail.

2.1.2.1 Structural SIMilarity Index

L'indice de similarité struturelle (SSIM, Structural SIMilarity) [43], est basé sur la mesure des similarités entre une image de référence et une image bruitée. Cette mérique considère que la dégradation à l'intérieur d'une image provient de la variation de l'information structurelle perçue. L'indice SSIM est une métrique FR évaluant la qualité d'image qui est basée sur les caractéristiques psychophysiques du Système Visuel Humain (HVS) [16] [42]. La figure 2.2 illustre le fonctionnement de cette métrique.

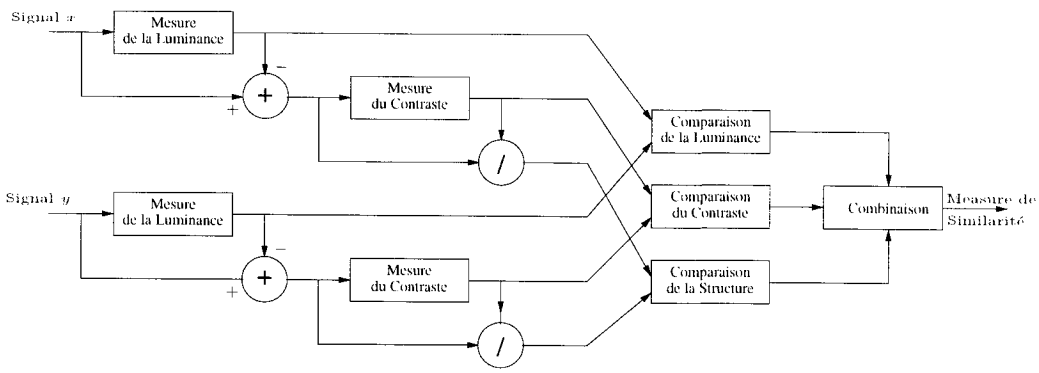


Figure 2.2 Diagramme décrivant le fonctionnement de la métrique SSIM.

L'indice SSIM combine les indices de comparaisons provenant de la luminance (voir équation (2.4)), du contraste (voir l'équation (2.5)), et de la structure (voir

l'équation (2.6)), d'une image. Cette métrique compare une image de référence x avec une image distortionnée y afin d'évaluer une mesure de similarité structurelle.

$$SSIM(x,y) = [l(x,y)]^{\alpha} \cdot [c(x,y)]^{\beta} \cdot [s(x,y)]^{\gamma}, \tag{2.3}$$

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \tag{2.4}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2\sigma_y^2 + C_2}, \tag{2.5}$$

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \tag{2.6}$$

où $C_1 = (K_1L)^2$, et $C_2 = (K_2L)^2$. Basé sur [43], nous adoptons $\alpha = \beta = \gamma = 1$, $C_3 = C_2/2$, $K_1 = 0.01$, et $K_2 = 0.03$. L représente la valeur dynamique d'un pixel, et puisqu'un signal monochrome de 8-bits/pixel est utilisé, $L = 255$. Les variables μ_x et μ_y représentent l'intensité moyenne des images x et y, et les variables σ_x et σ_y représentent la déviation standard, i.e. la racine carrée de la variance, des images x et y.

L'indice SSIM est calculé localement pour chaque pixel à l'aide d'une fonction de pondération circulaire symétrique Gaussienne, utilisant une matrice 11x11. Les indices SSIM calculés pour chaque position sont combinés afin de produire une image représentant une carte d'indice SSIM. La moyenne de la carte des indices SSIM est utilisée afin d'évaluer le valeur globale de la qualité de l'image évaluée:

$$MSSIM(X,Y) = \frac{1}{M}\sum_{j=1}^{M} SSIM(x,y). \tag{2.7}$$

où M représente le nombre de pixels dans l'image. Un code source C/C++ qui implémente cette métrique se trouve en référence à l'annexe C.

2.2 Différence entre validation formelle et vérification fonctionnelle

Il existe une grande ambiguïté pour ce qui a trait à la différence entre les définitions et les rôles de la vérification fonctionnelle et de la validation dans le domaine de la microélectronique. Comme le chapitre 3 porte sur la validation de modules de traitement vidéo, différencier ces deux activités demeure donc important. La validation représente l'ensemble des activités réalisées afin d'assurer la cohérence et la qualité des spécifications [3], i.e. les activités qui tentent de comparer les requis avec le modèle provenant d'une spécification textuelle ou exécutable. La validation permet de détecter les erreurs à l'intérieur d'une spécification et les activités liées à la validation se déroule lors de la spécification du modèle. Par contre, la vérification représente l'ensemble des activités réalisées afin de montrer la conformité d'une réalisation vis-à-vis de ses spécifications [34]. Un système de vérification n'est pas en mesure de détecter une erreur provenant des spécifications, contrairement à la validation. Un tel système vise plutôt à détecter les erreurs fonctionnelles à l'intérieur d'un design. Les activités reliées à la vérification se déroulent à la suite de la validation de la spécification du modèle.

2.3 Algorithmes de traitement vidéo

Afin de valider la méthodologie de validation de modules de traitement vidéo, présentée dans le chapitre 3, ce mémoire utilise des algorithmes de traitement vidéo provenant de deux classes d'applications, soit les algorithmes réducteur de bruits et les algorithmes de dé-entrelacement d'images. La section 2.3.1 présente les méthodes de réduction du bruit à l'intérieur d'une image, alors que la section 2.3.2 présente les méthodes de dé-entrelacement vidéo.

2.3.1 Méthodes de réduction du bruit vidéo

En général, le but d'un réducteur de bruit dans le domaine du traitement vidéo vise à améliorer le rendu visuel de l'image pour un observateur humain [29]. Les filtres réducteurs de bruit vidéo peuvent être divisés en trois catégories différentes [23], soit les filtres spatiaux, les filtres temporels et les filtres spatio-temporels. La section 2.3.1.1 présente les types de bruit vidéo expérimentés dans ce mémoire. Par la suite, les sections 2.3.1.2, 2.3.1.3, et 2.3.1.4 présentent respectivement le filtre Hybride, le filtre SUSAN, ainsi que le filtre de Wiener qui sont tous les trois des filtres réducteur de bruit vidéo faisant partie de la catégorie des filtres spatiaux.

2.3.1.1 Types de bruit vidéo

Trois types de bruit furent implémentés afin d'observer les performances des algorithmes réducteur de bruits vidéo, soit le bruit gaussien, le bruit sel et poivre, et le bruit dû à la compression d'image. Ces types de bruit furent sélectionnés parce qu'ils sont grandement retrouvés dans les applications de traitement d'images. Afin d'observer l'effet de chacun des types de bruit, l'image *Lena*, présentée à la figure 2.3, sera utilisée à titre d'exemple.

Le premier bruit considéré, soit le bruit gaussien, est souvent ajouté à l'intérieur d'une image durant une transmission analogique perturbée par du bruit aléatoire ou des interférences électromagnétiques. Ce mémoire utilise la méthode Box-Muller [4], décrite par l'équation (2.8), afin de générer des valeurs aléatoires de bruit gaussien. La valeur du bruit est ajoutée à chacun des pixels de l'image.

$$n = \sigma\sqrt{-2\ln a}\cos(2\pi b), \tag{2.8}$$

Figure 2.3 Image *Lena* ne contenant pas de bruit.

où a et b sont deux variables aléatoires uniformes indépendantes comprises dans l'intervalle $]0,1[$. Le niveau de la variance du bruit gaussien est contrôlé par la variable dénotée σ. La figure 2.4 illustre trois exemples de l'image *Lena* contenant du bruit gaussien avec différents niveaux de variance.

(a) (b) (c)

Figure 2.4 Image *Lena* contenant du bruit gaussien. (a) $\sigma = 0,01$, MSSIM = 0,629718; (b) $\sigma = 0,02$, MSSIM = 0,522856; (c) $\sigma = 0,05$, MSSIM = 0,384799.

Le deuxième bruit considéré, soit le bruit sel et poivre, apparaît comme des points noirs et blancs distribués aléatoirement à l'intérieur de l'image. Ce type de bruit est souvent causé par le processus de capture de l'image. Le niveau de sévérité de

ce type de bruit dans l'image est déterminé par une fraction de pixels, exprimée en pourcentage, affectée par l'algorithme. La figure 2.5 illustre trois exemples de l'image *Lena* contenant du bruit sel et poivre avec différents pourcentages de bruit.

(a) (b) (c)

Figure 2.5 Image *Lena* contenant du bruit sel et poivre. (a) 10% de bruit, MSSIM = 0,804710; (b) 20% de bruit, MSSIM = 0,664750; (c) 50% de bruit, MSSIM = 0,409215.

Le troisième et dernier type de bruit considéré, soit le bruit de compression, apparaît dans l'image sous forme d'effet de bloc suite à l'application d'un algorithme de compression d'image. Pour les besoins de ce mémoire, l'outil de compression JPEG fut utilisé afin de créer une détérioration réaliste [41]. L'outil de compression JPEG fut donc utilisé afin d'obtenir des images bruitées. Le niveau de sévérité de ce type de bruit est directement lié à la qualité de la compression de l'image [21] généré par l'outil JPEG. Plus le niveau de qualité est élevé, moins l'image est compressée, i.e. contient moins de bruit de compression. La figure 2.6 illustre trois exemples de l'image *Lena* contenant du bruit de compression avec différents niveaux de qualité.

(a) (b) (c)

Figure 2.6 Image *Lena* contenant du bruit de compression. (a) niveau de qualité = 35, MSSIM = 0,879585; (b) niveau de qualité = 15, MSSIM = 0,817463; (c) niveau de qualité = 10, MSSIM = 0,777333.

2.3.1.2 Algorithme du filtre Hybride

L'algorithme du filtre Hybride [29] provient de la combinaison du filtre Minimum Mean Square Error (MMSE) [26] et du filtre Sigma [24]. L'image filtrée g^*, à la sortie du filtre Hybride, est calculée comme suit:

$$g^*(x,y) = \mu_g + K_g(g(x,y) - \mu_g), \tag{2.9}$$

où K_g est un gain non-linéaire et μ_g est la moyenne locale autour de chaque pixel:

$$K_g = MAX\left(0, \frac{\sigma_g^2 - \sigma_n^2}{\sigma_g^2}\right), \tag{2.10}$$

$$\sigma_g^2 = \frac{1}{MN}\sum_{i,j\epsilon\eta}(W_g(g(x-i,y-j)) - \mu_g)^2, \tag{2.11}$$

$$\mu_g = \frac{1}{MN}\sum_{i,j\epsilon\eta} W_g(g(x-i,y-j)), \tag{2.12}$$

$$W_g = \left\{ \begin{array}{ll} 0 & \text{if } |g(x-i,y-j) - g(x,y)| \geq T_s \\ 1 & \text{if } |g(x-i,y-j) - g(x,y)| < T_s \end{array} \right\}, \tag{2.13}$$

où σ_g^2 représente la variance locale autour de chaque pixel, W_g représente le masque

adaptatif pour la segmentation à l'intérieur du filtre, et η représente la matrice N-par-M du voisinage local de chaque pixel se trouvant dans l'image bruitée g. Pour les besoins de ce mémoire et afin de réduire le temps de calcul, nous avons choisi $M = N = 7$. La variance du bruit décrit par σ_n^2 dans l'équation (2.11) et le seuil de segmentation décrit par T_s dans l'équation (2.13) sont deux paramètres devant être estimés afin d'utiliser cet algorithme. Le code source C/C++ du filtre Hybride se trouve en référence à l'annexe D dans la section D.1 et la figure 2.7 illustre le fonctionnement de l'algorithme.

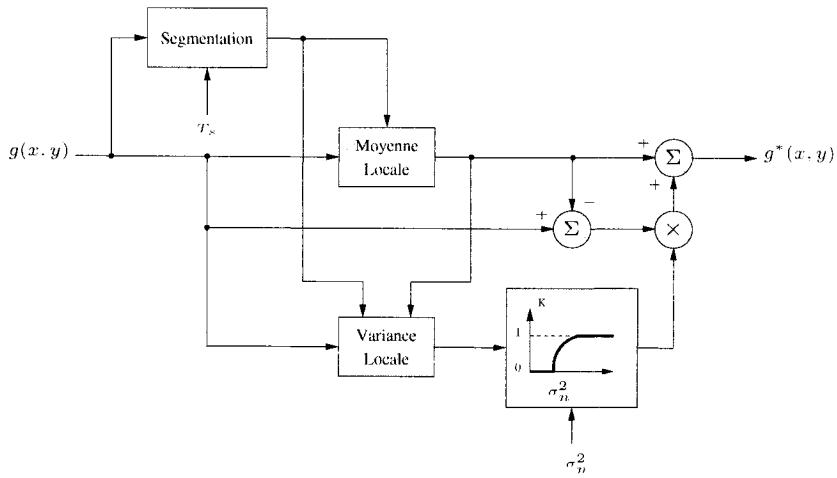


Figure 2.7 Schéma illustrant le fonctionnement de l'algorithme du filtre Hybride.

2.3.1.3 Algorithme du filtre SUSAN

Le filtre spatial Smallest Univalue Segment Assimilating Nucleus (SUSAN) [36] [37] est un filtre adaptatif préservant la structure d'une image en appliquant un lissage par rapport au voisinage faisant partie de la même région qu'un pixel. Ce filtre adaptatif peut-être décrit par l'équation:

$$J(x,y) = \sum_{i \neq 0, j \neq 0} I(x+i, y+j) \cdot S(i,j), \tag{2.14}$$

où

$$S(i,j) = \frac{e^{\frac{i^2+j^2}{2\tau^2} - \frac{(I(x+i,y+j)-I(x,y)^2}{\beta^2}}}{\sum_{i\neq 0, j\neq 0} e^{\frac{i^2+j^2}{2\tau^2} - \frac{(I(x+i,y+j)-I(x,y))^2}{\beta^2}}}, \tag{2.15}$$

et où J représente l'image filtrée et I représente l'image bruitée. L'algorithme réducteur de bruit SUSAN est contrôlé à l'aide de deux paramètres. Le premier paramètre, dénoté τ dans l'équation (2.15), contrôle la résolution spatiale qui a un effet sur le lissage de l'image filtrée. Le second paramètre, dénoté β dans l'équation (2.15), fixe le seuil d'intensité de luminosité. Le code source C/C++ du filtre SUSAN se trouve en référence à l'annexe D dans la section D.2

2.3.1.4 Filtre adaptatif de Wiener

Le filtre adaptatif de Wiener est un estimateur linéaire minimisant la racine carrée de la moyenne de l'erreur entre une image estimée et une image originale [17]. Lorsque $\sigma_n^2 < \sigma_j^2$ ou bien que $\sigma_j^2 \neq 0$, l'image filtrée F par le filtre de Wiener est calculée comme suit:

$$F(k,l) = \mu_j + \frac{\sigma_j^2 - \sigma_n^2}{\sigma_j^2}(G(k,l) - \mu_j), \tag{2.16}$$

où σ_n^2 représente la variance du bruit, σ_j^2 et μ_j représentent respectivement la variance et la moyenne locale autour d'un pixel j:

$$\sigma_n^2 = \frac{1}{S}\sum_{j=1}^{S}\sigma_j^2, \tag{2.17}$$

$$\sigma_j^2 = \frac{1}{NM}\sum_{k,l\epsilon\eta} G(k,l)^2 - \mu_j^2, \tag{2.18}$$

$$\mu_j = \frac{1}{NM}\sum_{k,l\epsilon\eta} G(k,l), \tag{2.19}$$

où S représente la taille de l'image, et η représente la matrice N-par-M du voisinage local autour de chaque pixel de l'image bruitée G. Par contre, lorsque $\sigma_n^2 \geq \sigma_j^2$ ou bien que $\sigma_j^2 = 0$, le filtre de Wiener retourne $F(k,l) = \mu_j$. Ce mémoire utilise les valeurs suivantes pour les paramètres de cet algorithme: $N = 3$; $M = 3$. Le code source C/C++ du filtre de Wiener se trouve en référence à l'annexe D dans la section D.3 et la figure 2.8 illustre le fonctionnement de l'algorithme.

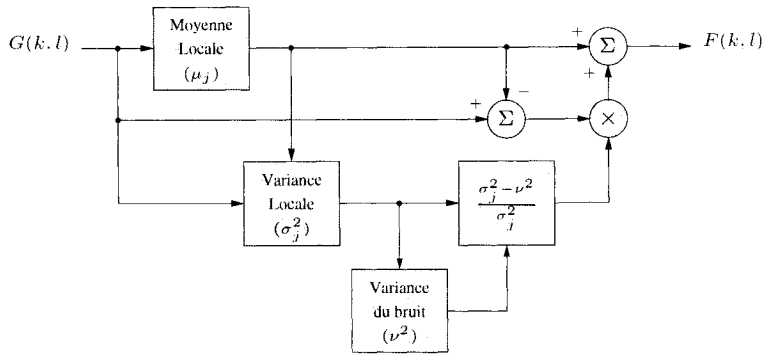


Figure 2.8 Schéma illustrant le fonctionnement de l'algorithme du filtre Wiener.

2.3.2 Méthodes de dé-entrelacement vidéo

Le dé-entrelacement est un processus important qui convertit des séquences pour lesquelles le balayage est entrelacé en séquences pour lesquelles le balayage est progressif. La figure 2.9 illustre la tâche qu'effectue le dé-entrelacement vidéo [18]. Les champs à l'entrée du signal vidéo, contenant les échantillons des lignes verticales paires ou impaires d'une image, doivent être convertis en des images entières. Ces dernières représentent les mêmes images correspondant aux champs à l'entrée, mais elles contiennent tous les échantillons des lignes de l'image. Certains effets visuels incomfortables pour l'oeil humain peuvent apparaître lors d'un mauvais dé-entrelacement. Fondamentalement, les algorithmes de dé-entrelacement peuvent être caractérisés par quatre catégories [27]: le dé-entrelacement intra-champ, le dé-entrelacement inter-champ, le dé-entrelacement adaptatif au mouvement, et le

dé-entrelacement avec compensation de mouvement.

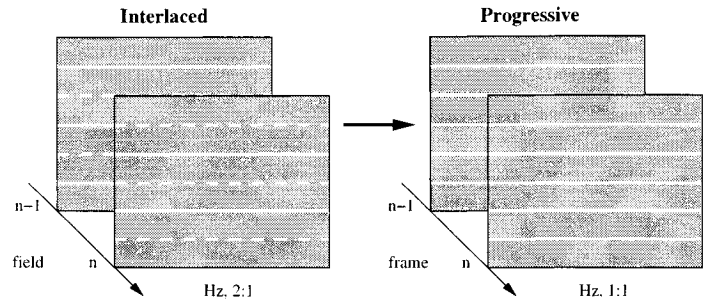


Figure 2.9 Schéma illustrant la tâche rattaché au dé-entrelacement tiré de [33].

Les algorithmes de dé-entrelacement de la catégorie intra-champ sont plus efficaces pour des séquences contenant beaucoup de mouvement, alors que les algorithmes de la catégorie inter-champ sont plus efficaces pour les séquences contenant plusieurs régions statiques dans l'image. Par contre, les algorithmes de dé-entrelacement de la catégorie adaptatif au mouvement combinent les avantages des algorithmes de dé-entrelacement intra-champ et inter-champ. Ils utilisent un algorithme de détection de mouvement appliquant l'algorithme de type intra-champ dans le cas d'une région avec mouvement, ce qui permet d'éviter les artéfacts spatiaux à l'intérieur de l'image. L'algorithme de type inter-champ est appliqué dans le cas de la détection d'une région statique, ce qui permet d'éviter une perte de résolution [33]. Finalement, les algorithmes de dé-entrelacement du type compensation de mouvement possèdent un plus grand potentiel de produire de meilleurs résultats, mais ces derniers sont plus complexes par le fait qu'ils requièrent un module d'estimation de mouvement très exigeant en terme de puissance de calcul. La section 2.3.2.1 présente les détails du fonctionnement d'un algorithme du type intra-champ qui fut sélectionné parce que les détails mathématiques de son implémentation sont disponibles dans la littérature.

2.3.2.1 Algorithme de dé-entrelacement ELA modifié

L'algorithme Edge-based Line Average (ELA) modifié [22] [27] est un algorithme de dé-entrelacement intra-champ largement utilisé dans le domaine du traitement vidéo. Un schéma illustrant les différentes étapes de cet algorithme afin d'interpoler les pixels des lignes manquantes, dénotés Z_{ELA}, est présenté à la figure 2.10. Les variables contenues dans ce schéma sont décrites plus loins à l'intérieur de cette section.

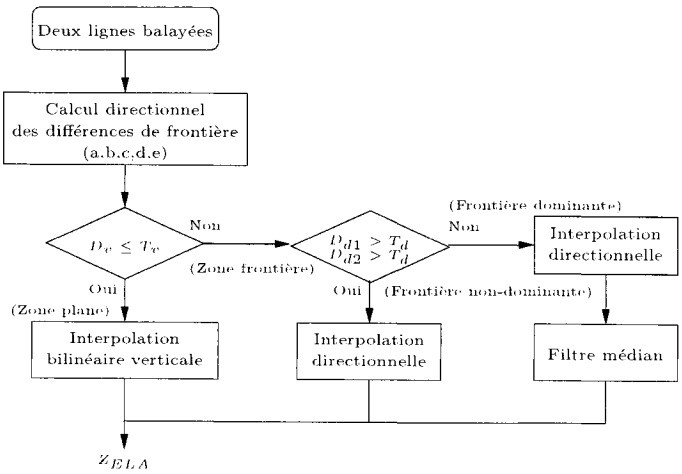


Figure 2.10 Schéma du fonctionnement de l'algorithme ELA modifié.

L'algorithme reconstruit la ligne k à partir des lignes $k-1$ et $k+1$. Dans la figure 2.11, a, b, c, d, et e représentent les différences directionnelles autour du pixel $x(k,n)$ qui doit être interpolé. Ces cinq différences directionnelles sont définies par:

$$a = |x(k-1,n-2) - x(k+1,n+2)| \tag{2.20}$$

$$b = |x(k-1,n-1) - x(k+1,n+1)| \tag{2.21}$$

$$c = |x(k-1,n) - x(k+1,n)| \tag{2.22}$$

$$d = |x(k-1,n+1) - x(k+1,n-1)| \tag{2.23}$$

$$e = |x(k-1,n+2) - x(k+1,n-2)| \tag{2.24}$$

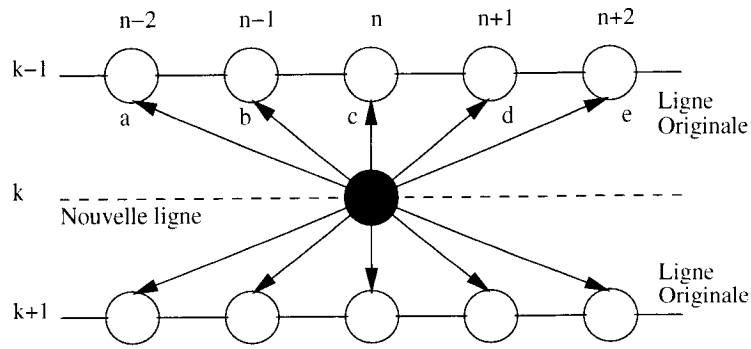


Figure 2.11 Patron de pixels utilisé dans l'algorithme ELA modifié.

La plus petite différence, parmis les cinq différences directionnelles, représente la meilleure corrélation. Les équations des interpolations directionnelles sont définies par:

$$x(k,n) = \begin{cases} \frac{x(k-1,n-2)+x(k+1,n+2)}{2} & \text{si } min(a,b,c,d,e) = a \\ \frac{x(k-1,n-1)+x(k+1,n+1)}{2} & \text{si } min(a,b,c,d,e) = b \\ \frac{x(k-1,n)+x(k+1,n)}{2} & \text{si } min(a,b,c,d,e) = c \\ \frac{x(k-1,n+1)+x(k+1,n-1)}{2} & \text{si } min(a,b,c,d,e) = d \\ \frac{x(k-1,n+2)+x(k+1,n-2)}{2} & \text{si } min(a,b,c,d,e) = e \end{cases} \tag{2.25}$$

Par la suite, l'algorithme calcule si la direction estimée est une frontière dominante ou non. Afin de déterminer si une direction estimée est une frontière non-dominante, l'algorithme utilise les différences entre la différence directionnelle provenant de la meilleur corrélation et les différences directionnelles opposées à la meilleur corrélation. Ces différences sont dénotées D_{d1} et D_{d2}:

$$\text{si } min(a,b,c,d,e) = a \quad \begin{cases} D_{d1} = |a-d| \\ D_{d2} = |a-e| \end{cases} \tag{2.26}$$

$$\text{si } min(a,b,c,d,e) = b \quad \begin{cases} D_{d1} = |b-d| \\ D_{d2} = |b-e| \end{cases} \tag{2.27}$$

$$\text{si } min(a,b,c,d,e) = d \quad \begin{cases} D_{d1} = |d-a| \\ D_{d2} = |d-b| \end{cases} \tag{2.28}$$

$$\text{si } min(a,b,c,d,e) = e \quad \begin{cases} D_{d1} = |e-a| \\ D_{d2} = |e-b| \end{cases} \tag{2.29}$$

L'algorithme de dé-entrelacement ELA modifié utilise deux paramètres dénoté T_v et T_d. Le paramètre T_v représente le seuil vertical utilisé pour la différence directionnelle dénotée c dans la figure 2.11. Le paramètre T_d représente le seuil de différence directionnelle utilisé pour les deux différences D_{d1} et D_{d2}, représentées par les équations (2.26), (2.27), (2.28), et (2.29), afin de déterminer si la région est une frontière dominante. Pour terminer, le calcul du filtre médian, se trouvant dans le schéma du fonctionnement de l'algorithme ELA modifié à la figure 2.10, est décrit par l'équation (2.30).

$$Z_{ELA} = Median[x(k,n-1), x(k,n+1), x(k,n)] \tag{2.30}$$

2.4 Conclusion

Dans ce chapitre, les différents concepts et aspects concernant la méthodologie présentée dans le Chapitre 3 de ce Mémoire furent élaborées. Cette méthodologie concerne la validation automatique d'algorithme de traitement vidéo. Comme il est expliqué à la section 2.2, les activités reliées à la validation se déroulent lors de la spécification du modèle et permettent de détecter les erreurs à l'intérieur de celle-ci, ce que la vérification fonctionnelle ne peut accomplir. Une métrique objective mesurant la qualité d'image, nommée indice MSSIM, fut sélectionnée afin de permettre le retrait d'observateur humain de la tâche d'évaluation de la qualité d'image à l'intérieur de la méthodologie de validation présentée dans ce mémoire. Pour terminer, les algorithmes de traitement vidéo, provenant de la classe d'application

des algorithmes réducteur de bruit et de la classe d'application des algorithmes de dé-entrelacement, furent décrits à la section 2.3. Trois filtres numériques d'image ainsi qu'un algorithme de dé-entrelacement vidéo furent sélectionnés afin de valider la méthodologie de validation automatique d'algorithme de traitement vidéo proposée et présentée au chapitre 3. Il est à noter que la rétine de l'oeil humain est composée de 100 million de cellules appelés *bâtonnets*, sensibles à l'intensité lumineuse, et de 5 million de cellules appelés *cônes*, sensibles á trois longueurs d'onde de lumière, soit approximativement les couleur rouge, bleue et verte [33]. Les algorithmes de traitement vidéo sont généralement appliqués dans le domaine de couleur YUV, où le signal Y représente la luminance et les signaux U et V représentent la chrominance. Comme l'oeil humain est beaucoup plus sensible à la variation de la luminosité, les algorithmes de traitement vidéo sont appliqués seulement sur le signal de la luminance. Par contre, dans le cas du dé-entrelacement vidéo, une méthode nommée duplication de lignes est appliquée pour les signaux de la chrominance [19].

CHAPITRE 3

VALIDATION DE MODULES DE TRAITEMENT VIDÉO

Les architectures de modules de traitement vidéo sont généralement dérivées à partir d'algorithmes validés antérieurement utilisant une implémentation C/C++ ou MATLAB/SIMULINK. Elles sont généralement simulées en résolution virgule-flottante qui est une résolution très coûteuse en termes de coût matériel. Une bonne approche afin de minimiser ou d'optimiser la complexité d'une implémentation matérielle, sa performance et sa consommation de puissance, est l'utilisation de la précision en virgule-fixe. De plus, certains algorithmes de traitement vidéo contiennent des paramètres ayant besoin d'être fixés afin d'obtenir une performance optimale. Ces paramètres sont généralement déterminés empiriquement.

La traduction d'une formulation en virgule-flottante à une formulation en virgule-fixe, pour une implémentation matérielle, peut causer quelques problèmes. Par exemple, la précision de chaque opérande et la valeur de chaque paramètre contenu dans un algorithme doivent être minutieusement sélectionnées afin de préserver la stabilité de l'algorithme, ainsi que l'acuité de son traitement. De plus, cette sélection peut détériorer le comportement d'un algorithme de traitement vidéo en produisant des artéfacts visuels déplaisant pour un observateur humain.

Une nouvelle méthode systématique et automatique est présentée afin de valider des algorithmes de traitement vidéo. Le but de cette méthode est de pousser à un niveau d'abstraction plus élevé le processus de validation d'algorithmes de traitement vidéo pour un concepteur matériel. Afin d'atteindre ce but, la nouvelle méthode proposée utilise une plateforme de validation automatique d'algorithmes

de traitement vidéo prenant avantage de nouvelles métriques objectives mesurant la qualité des images. Le fait d'utiliser des métriques objectives mesurant la qualité des images permet de retirer un observateur humain du processus d'évaluation de cette qualité ou d'en augmenter sa productivité.

Ce chapitre est divisé en trois sections principales. La section 3.1 présente la plateforme de validation d'algorithmes de traitement vidéo. Un article de revue [10], concernant cette section, est disponible à l'annexe A. Par la suite, la section 3.2 présente les modifications apportées à la plateforme de validation afin d'estimer automatiquement des paramètres pour un algorithme de traitement vidéo. Un article de conférence [11], concernant cette section, est disponible à l'annexe B. Pour terminer, la section 3.3 formule les conclusions de ce chapitre.

3.1 Plateforme de validation d'algorithmes de traitement vidéo

Le passage d'un algorithme de traitement vidéo en résolution virgule-flottante vers une implémentation en résolution virgule-fixe, une étape du processus qui conduit à une implémentation matérielle, peut causer certains problèmes. Par exemple, la précision de chaque opérande devrait être méticuleusement choisie afin de préserver la stabilité et l'acuité du comportement de l'algorithme. La sensibilité de l'effet que peut avoir la taille de mots d'une opérande, à l'intérieur d'un algorithme de traitement vidéo, n'est pas très claire *a priori*. De plus, ce passage peut détériorer le comportement d'un algorithme de traitement vidéo en produisant des artéfacts visuels déplaisant pour un observateur humain.

La validation aide un concepteur de matériel à concevoir et à implémenter des systèmes complexes en assurant, avec un haut niveau de confiance, que ces derniers sont conformes dans toutes les circonstances [13]. La technique la plus commune

en validation de systèmes est basée sur les tests et les simulations. Des techniques plus formelles, comme le *model checking*, développées afin d'explorer de manière exhaustive toutes les possibilités de comportement à l'intérieur d'un système, sont applicables pour la validation de la partie de contrôle d'un système, mais ne sont pas applicables pour la validation des chemins de données d'un système. En effet, ces techniques utilisent une logique booléenne temporelle afin de valider les propriétés d'un modèle provenant d'une spécification exécutable. Cependant, ces techniques ne permettent dans aucun cas de valider la qualité d'une image produite par l'implémentation d'un algorithme de traitement vidéo.

La méthode introduite dans [6] permet d'optimiser automatiquement les chemins de données à l'intérieur d'une implémentation d'un algorithme de traitement vidéo. Cette méthode exploite un outil de détermination automatique de la taille de mots (AWLDT) [7] [8] qui fut choisi pour deux raisons principales. Tout d'abord, cet outil fut développé au sein du groupe de recherche en microélectronique (GRM) de l'École Polytechnique de Montréal et son implémentation était donc disponible. Par la suite, cet outil est le seul, se trouvant dans la littérature, pouvant déterminer automatiquement la taille de mots à l'aide de simulations. Cette méthode fut raffinée afin de faciliter la validation orientée performance applicable à l'implémentation matérielle d'algorithmes de traitement vidéo. En effet, les expériences conduites à l'aide d'une première méthode ont démontré que dépendamment de la métrique utilisée, celle-ci peut produire des anomalies dans les images résultantes. C'est ce qui a motivé le développement d'une méthode plus robuste qui évalue automatiquement la performance d'algorithmes de traitement vidéo tout en produisant des images valides, et cela, sans l'intervention d'un observateur humain qui aurait pour tâche de valider les résultats. La composante clée de cette nouvelle méthodologie repose sur la combinaison de l'outil AWLDT avec une nouvelle métrique pouvant rejeter des solutions générant des images distortionnées. Les composants dont se

compose la plateforme de validation d'algorithmes de traitement vidéo, développés afin d'accomplir cette tâche, sont décrits plus en profondeur dans la section 3.1.1.

3.1.1 Composantes de la plateforme

La plateforme de validation d'algorithmes de traitement vidéo, illustrée à la figure 3.1, utilise les mêmes trois étapes que le *model checking* [13], i.e. la modélisation, la spécification et la vérification, même si la méthodologie employée est très différente. Lors de la première étape, le concepteur de matériel doit produire un modèle de haut niveau de l'algorithme en langage C/C++ ou SystemC. Ce modèle doit être modifié en une représentation Control Data Flow Graph (CDFG) afin de déterminer quelles seront les opérandes à analyser dans l'algorithme de traitement vidéo. Lors de la deuxième étape, le concepteur de matériel doit définir des objectifs de performances qui deviendront éventuellement des propriétés de la spécification du modèle. L'objectif de performance de notre application, qui est décrite dans la section 3.1.1.2, utilise des métriques objectives de mesure de la qualité d'images, contrairement à la validation classique qui utilise de la logique temporelle, e.g. la logique temporelle linéaire (LTL) ou les arbres de calculs logiques (CTL), afin de décrire les propriétés de la spécification du modèle [13]. Lors de la troisième étape, le concepteur de matériel doit valider le fait que chaque objectif de performance est rencontré par le modèle. Afin de procéder à cette dernière étape, la plateforme de validation d'algorithmes de traitement vidéo utilise l'outils AWLDT, contrairement à l'utilisation d'un *model checker* pour la validation classique. À partir d'objectifs de performance et d'une banque d'images, l'outil AWLDT génère un modèle optimisé du chemin de données de l'algorithme qui rencontre les performances ciblées pour l'implémentation matérielle de l'algorithme de traitement vidéo.

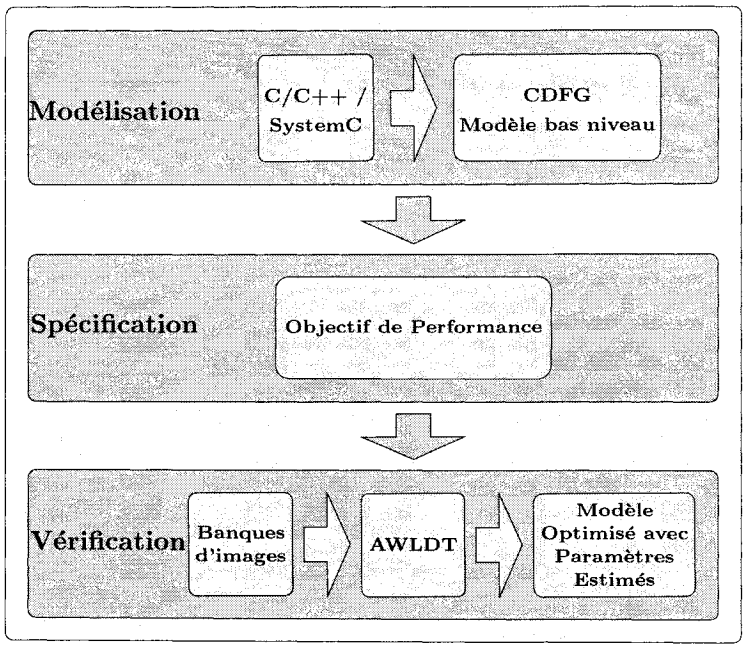


Figure 3.1 Diagramme du fonctionnement de la plateforme de validation d'algorithmes de traitement vidéo.

3.1.1.1 Représentation bas niveau de l'algorithme

Afin de déterminer les opérandes à analyser à l'intérieur de l'algorithme de traitement vidéo, il faut procéder à une conversion manuelle de la modélisation de l'algorithme. Il faut que le concepteur passe d'un modèle haut niveau C/C++, ou SystemC, vers un modèle bas niveau, en langage C, sous la forme d'un DFG ordonnancé [12] ou d'un CDFG [28]. La figure 3.2 illustre l'exemple du DFG ordonnancé pour le calcul de la moyenne locale, dénoté μ_j dans l'Équation (2.18) du filtre de Wiener décrit à la section 2.3.1.4. Il est possible d'y observer que son DFG comporte 9 opérandes à analyser, soit 8 additions ainsi qu'une division.

Voici le code haut niveau C/C++, de l'exemple illustré à la figure 3.2, calculant la moyenne locale μ_j:

```
// Local Mean
```

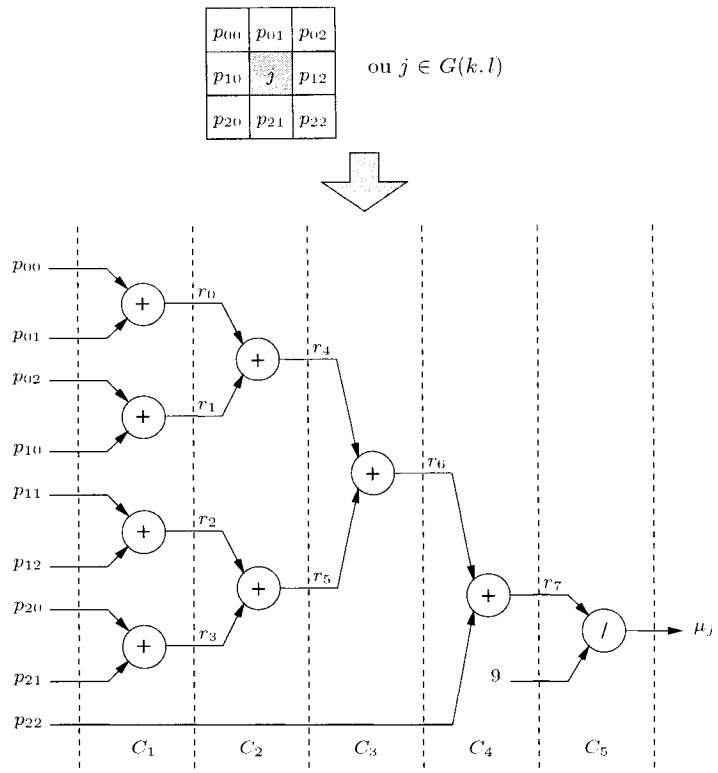


Figure 3.2 DFG ordonnancé de la moyenne locale μ_j pour le filtre Wiener (voir équation (2.18))

```
for(y = -1; y <= 1; y++){
  for(x = -1; x <= 1; x++){
    mu_j += G[l + y][k + x];
  }
}
mu_j = mu_j / 9;
```

Voici maintenant le code bas niveau en langage C, basé sur la représentation du DFG ordonnancé illustré à la figure 3.2, calculant la moyenne locale μ_j:

```
// Local Mean
// Cycle 1
R0 = P00 + P01;
```

```
R1 = P02 + P10;
R2 = P11 + P12;
R3 = P20 + P21;
// Cycle 2
R4 = R0 + R1;
R5 = R2 + R3;
// Cycle 3
R6 = R4 + R5;
// Cycle 4
R7 = R6 + P22;
// Cycle 5
mu_j = R7 / 9;
```

3.1.1.2 Métrique de performance

L'outil AWLDT, à l'aide d'une heuristique appliquant une descente de gradient, cherche la combinaison de tailles de mot $\{WL_i\}$ minimisant le coût de l'implémentation matérielle de l'algorithme de traitement vidéo. Afin d'effectuer sa recherche d'une solution optimale, l'outil AWLDT est guidé par une métrique de performance représentée par l'inverse d'un rapport (IRATIO). La métrique globale IRATIO, illustrée à la figure 3.3, calcule le rapport de la métrique de qualité d'image (OIQM) obtenue pour le chemin de données en précision finie (OIQM virgule-fixe) sur la même métrique obtenue pour la précision complète (OIQM virgule-flottante).

La métrique objective de mesure de qualité d'images (OIQM) utilisée dans la plateforme de validation d'algorithmes de traitement vidéo se nomme MSSIM et est décrite dans la section 3.1.1.2. La métrique IRATIO se calcule donc comme suit:

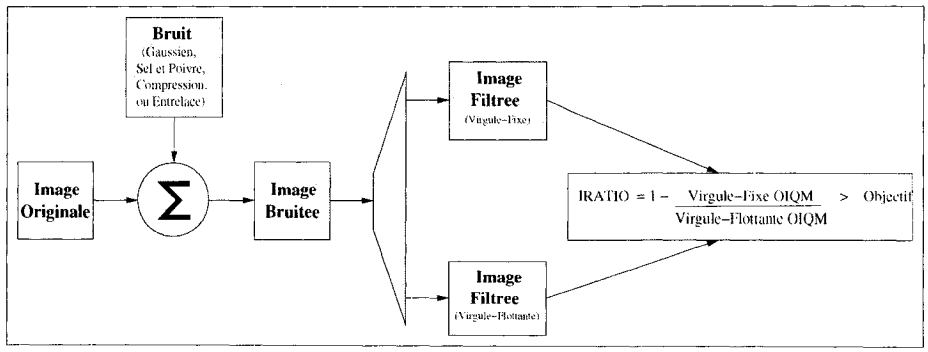


Figure 3.3 Diagramme de l'évaluation de l'objectif de performance.

$$IRATIO = \max_{m=0,M-1} \left\{ 1 - \frac{MSSIM(I_m^o, I_m^p)}{MSSIM(I_m^o, I_m^f)} \right\} \tag{3.1}$$

où M représente le nombre d'images à traiter, I_m^o représente la m^{ieme} image originale (I^o), I_m^p représente la m^{ieme} image filtrée en résolution virgule-fixe (I^p) par l'algorithme de traitement vidéo, I_m^f représente la m^{ieme} image filtrée en résolution virgule-flottante (I^f) par l'algorithme de traitement vidéo, et max représente la fonction maximum.

3.1.1.3 Fichier de simulation

Comme l'outil AWLDT est basé sur la simulation de l'algorithme afin de trouver les tailles optimales des opérandes, ce dernier a besoin d'un fichier de simulation. Ce fichier est généré par la plateforme de validation d'algorithmes de traitement vidéo à l'aide d'une banque d'images [47]. Pour chacune des images sélectionnées par le concepteur, le fichier contiendra la valeur de la métrique calculée pour l'image courante filtrée en résolution virgule-flottante, la taille de l'image courante, l'image courante de référence et l'image courante bruitée. Le type de bruit, ainsi que son niveau d'intensité, est sélectionné par le concepteur. Le fichier modèle C/C++ qui sert à simuler un algorithme réducteur de bruit dans l'outil AWLDT se trouve à

l'annexe E dans la section E.1. De plus, le fichier modèle C/C++ afin de calculer l'objectif de performance dans l'outil AWLDT se trouve dans la même section.

3.1.2 Anomalies observées

Afin de tester notre méthodologie de validation d'algorithmes de traitement vidéo, le filtre adaptatif de Wiener, décrit dans la section 2.3.1.4, fut sélectionné en tant que modèle de référence. À la suite de l'optimisation du chemin de données du filtre de Wiener en représentation virgule-fixe, à l'aide de la plateforme de validation d'algorithmes de traitement vidéo proposée, des résultats furent obtenus avec deux types d'anomalies. Certaines solutions optimisées, générées par l'outil AWLDT, contenaient des pixels négatifs, alors que d'autres contenaient des pixels ayant une intensité lumineuse qui divergeait totalement de celle de leurs voisins. Ces résultats furent obtenus même si la solution de l'implémentation en virgule-fixe de l'algorithme, trouvée par la plateforme de validation d'algorithmes de traitement vidéo, rencontrait les critères de performance sévères qui avaient été fixés.

Le tableau 3.1 présente les résultats, obtenus pour les métriques Mean Square Error (MSE), Peak Signal-to-Noise Ratio (PSNR) et pour l'indice MSSIM. Ces résultats sont rapportés pour neuf images différentes [47]. Chaque image originale contient du bruit gaussien avec un niveau de variance de 0,01. Les images bruitées furent filtrées à l'aide du filtre de Wiener en résolution virgule-flottante et virgule-fixe. L'implémentation du filtre de Wiener en virgule-fixe fut générée par la plateforme de validation d'algorithmes de traitement vidéo utilisant la métrique MSSIM ainsi qu'un objectif de performance de 0,95. L'implémentation optimisée en virgule-fixe du filtre de Wiener provenant de la plateforme de validation d'algorithmes de traitement vidéo génère des anomalies visuelles. La figure 3.4 illustre un exemple, provenant du tableau 3.1, d'une section de l'image *Lena* filtrée en résolution virgule-

fixe.

Tableau 3.1 Valeurs de métriques objectives mesurant la qualité globale d'image, pour différentes images contenant du bruit gaussien avec un niveau de variance de 0,01, ainsi que pour les images correspondantes filtrées à l'aide du filtre de Wiener en résolution virgule-flottante et virgule-fixe.

Image	Bruit Gaussien			Filtre Wiener (virgule-flottante)			Filtre Wiener (virgule-fixe)		
	MSE	PSNR	MSSIM	MSE	PSNR	MSSIM	MSE	PSNR	MSSIM
Arctichare	373,29	22,410	0,292278	103,76	27,971	0,732639	141,40	29,626	0,711137
Boat	189,18	25,362	0,644498	81,08	29,041	0,807659	97,51	28,240	0,795193
Cameraman	300,67	23,350	0,541062	112,06	27,636	0,739830	135,10	26,824	0,729231
Cat	207,38	24,963	0,755108	128,70	27,035	0,844317	147,08	26,455	0,832029
Couple	164,78	25,962	0,729978	116,78	27,457	0,767334	128,54	27,041	0,773001
Fruits	273,73	23,758	0,450619	69,81	29,692	0,766572	91,73	28,506	0,745399
Goldhill	145,21	26,511	0,753261	90,32	28,573	0,770426	110,31	27,705	0,754731
Lena	172,32	25,767	0,629845	66,37	29,911	0,811397	87,31	28,720	0,788909
Peppers	166,65	25,913	0,620603	48,42	31,281	0,868091	64,72	30,021	0,849463

Figure 3.4 Section de l'image *Lena* filtrée en résolution virgule-fixe provenant du tableau 3.1.

Même si l'objectif de performance est atteint pour la solution optimale de l'implémentation du filtre Wiener générée par l'outil AWLDT, celle-ci contient des anomalies visuelles et cette solution aurait due être rejetée. Les sections 3.1.2.1 et 3.1.2.2 élaborent plus en profondeur sur les deux types d'anomalies, générées par la plateforme de validation d'algorithmes de traitement vidéo, qu'il faut absolument éviter.

3.1.2.1 Pixels négatifs

L'optimisation du chemin de données génère des solutions contenant des opérandes optimisées qui propagent des résultats négatifs jusqu'à la sortie du filtre numérique. Comme la plateforme de validation d'algorithmes de traitement vidéo utilise une métrique objective calculée globalement, quelques pixels negatifs n'ont qu'une très faible influence sur le résultat final. Cela explique le fait qu'il est donc possible d'obtenir une solution satisfaisant le critère de performance fixé *a priori* et générant des images contenant ce type d'anomalie. Les images illustrées dans la figure 3.5 furent obtenues à partir de simulations du filtre Wiener fonctionnant en résolution virgule-fixe. Cette implémentation du filtre numérique rencontre un objectif de performance de 0,9832, qui est une valeur relativement élevée pour cette métrique, ce qui n'empêche pas l'apparition de ce type d'anomalie. L'image originale contient 85% de bruit de compression JPEG. Les pixels négatifs sont représentés par des pixels blancs dans l'image filtrée en virgule-fixe, illustrée à la figure 3.5(b), et sont rapportées par des taches noires dans la carte des indices SSIM, illustrée à la figure 3.5(c).

3.1.2.2 Pixels divergeant de leur voisin

Des résultats furent aussi obtenus avec des solutions optimisées contenant des pixels possédant un très bas niveau d'intensité lumineuse. Même un nombre restreint de pixels contenant ce type d'anomalie dans une image peuvent être détectés par un observateur humain. Cela est dû au fait que l'intensité lumineuse des pixels contenant cette anomalie diffère significativement de celle de leurs voisins. Les images contenues dans les figures 3.6, 3.7, et 3.8, obtenues à partir de la simulation de l'implémentation du filtre de Wiener provenant d'une représentation en virgule-fixe qui atteint un objectif de performance de 0,9832, illustre ce type d'anomalie.

(a) (b) (c)

Figure 3.5 Résultats de simulation contenant des pixels négatifs. (a) Image filtrée *Lena* en virgule-flottante, MSSIM = 0,834480; (b) Image filtrée *Lena* en virgule-fixe, MSSIM = 0,822677; (c) Carte des indices SSIM comparant l'image *Lena* filtrée en virgule-flottante et virgule-fixe.

Chaque image source contient du bruit gaussien avec un niveau de variance de 0,01. Les pixels divergeant de leurs voisins sont rapportés par des taches noires dans les cartes d'indices SSIM des figures 3.6(c), 3.7(c). et 3.8(c). Le fait que l'outil AWLDT accepte des solutions avec des pixels divergeant de leurs voisins peut être encore expliqué par l'utilisation d'une métrique calculée globalement.

3.1.2.3 Instabilités numériques

Ces anomalies, retrouvées dans les solutions optimales, peuvent être expliquées par le fait que le filtre de Wiener devient instable numériquement lorsque certaines opérandes sont trop optimisées. En particulier, dans l'implémentation du filtre de Wiener, le résultat d'une des opérandes consiste à calculer l'opération $G(k,l) - \mu_j$ dans l'Équation (2.16). Cette opération spécifique peut générer des résultats négatifs qui sont propagés vers trois autres opérandes dans le filtre. La figure 3.9 illustre la zone critique du chemin de données du filtre de Wiener et la propagation des valeurs négatives est représentée par des traits pointillés. Ces valeurs

(a) (b) (c)

Figure 3.6 Résultats de simulation obtenus avec des pixels divergeant de leurs voisins. (a) Image filtrée *Boat* en virgule-flottante, MSSIM = 0,807659; (b) Image filtrée *Boat* en virgule-fixe, MSSIM = 0,776457; (c) Carte des indices SSIM comparant l'image *Boat* filtrée en virgule-flottante et virgule-fixe.

négatives peuvent se présenter autant en résolution virgule-flottante que virgule-fixe. Cependant, le résultat de cette opération sert de numérateur à l'intérieur d'une opération effectuant une division. Si la valeur du dénominateur est trop tronquée en résolution virgule-fixe, l'opération effectuant la division génère un résultat plus grand qu'en temps normal. La dernière opération consiste à additionner ce résultat négatif, possiblement amplifié, avec la variance locale du pixel courant, ce qui explique la génération de pixels négatifs. Ces anomalies dans les images résultant de l'implémentation virgule-fixe nous ont motivés à développer une nouvelle métrique objective mesurant la qualité d'image (OIQM). Cette OIQM est présentée dans la section 3.1.3 et elle est capable de détecter automatiquement des instabilités numériques à l'intérieur d'un algorithme de traitement vidéo calculée en virgule-fixe.

(a) (b) (c)

Figure 3.7 Résultats de simulation obtenus avec des pixels divergeant de leurs voisins. (a) Image filtrée *Cameraman* en virgule-flottante, MSSIM = 0,739830; (b) Image filtrée *Cameraman* en virgule-fixe, MSSIM = 0,711923; (c) Carte des indices SSIM comparant l'image *Cameraman* filtrée en virgule-flottante et virgule-fixe.

3.1.3 Utilisation d'un seuil

Une première solution fut utilisée afin de résoudre l'anomalie présentant des pixels négatifs illustrée à la figure 3.5. Tous les résultats produit par l'outil AWLDT contenant des pixels négatifs devaient être automatiquement rejetés. Malgré cette nouvelle mesure, des résultats contenant le deuxième type d'anomalie, soit des pixels ayant une intensité lumineuse divergeant de celle de leurs voisins, furent quand même obtenus (voir figure 3.8). Afin de remédier à l'anomalie des pixels divergeant de leurs voisins, la métrique MSSIM fut complémentée par un test additionnel basé sur une variable nommée *Pixel Threshold* (PT). Ce test fixe un seuil pour la valeur absolue maximale de la différence entre l'intensité lumineuse des pixels produits par l'implémentation en virgule-flottante et celle en virgule-fixe.

Le seuil *Pixel Threshold* est une métrique non-linéaire locale qui permet de rejeter une solution si un seul pixel de l'image ne satisfait pas le critère de performance. La nouvelle OIQM, générée à partir d'une métrique globale et d'une métrique locale,

(a) (b) (c)

Figure 3.8 Résultats de simulation obtenus avec des pixels divergeant de leurs voisins. (a) Image filtrée *Lena* en virgule-flottante, MSSIM = 0,811397; (b) Image filtrée *Lena* en virgule-fixe, MSSIM = 0,775641; (c) Carte des indices SSIM comparant l'image *Lena* filtrée en virgule-flottante et virgule-fixe.

enlève les instabilités numériques à l'intérieur des implémentations matérielles optimisées d'algorithme de traitement vidéo. Les figures 3.10, 3.11, et 3.12 illustrent les résultats obtenus à partir de la simulation du filtre de Wiener en résolution virgule-fixe répondant à un objectif de performance de 0,95 et ayant un seuil PT maximal de 34. Chaque image originale contient du bruit gaussien avec un niveau de variance de 0,01. Les images produites par l'implémentation du filtre de Wiener en résolution point-fixe, validée à l'aide de la nouvelle OIQM, ne contiennent pas

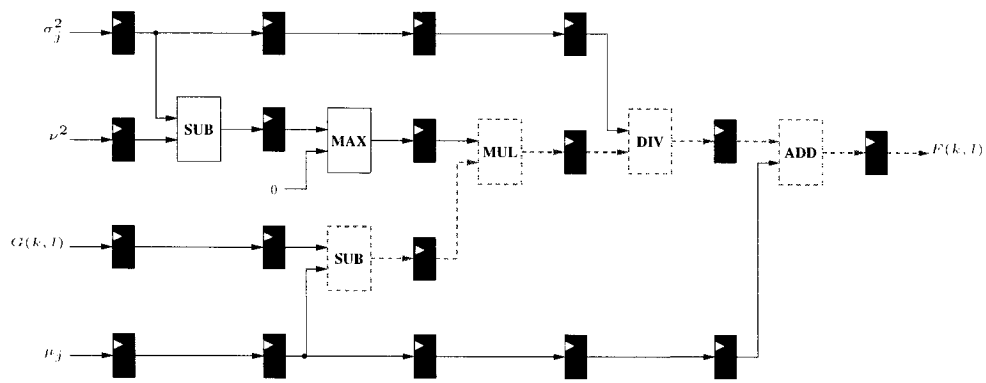


Figure 3.9 Zone critique du chemin de données du filtre de Wiener.

d'anomalie, même si l'objectif global de performance est plus bas que celui utilisé pour produire les résultats présentés aux figures 3.6, 3.7, et 3.8.

(a) (b) (c)

Figure 3.10 Résultats de simulation obtenus avec la nouvelle OIQM. (a) Image filtrée *Boat* en virgule-flottante, MSSIM = 0,807659; (b) Image filtrée *Boat* en virgule-fixe, MSSIM = 0,795193; (c) Carte des indices SSIM comparant l'image *Boat* filtrée en virgule-flottante et virgule-fixe.

La figure 3.13 illustre la même section de l'image *Lena* filtrée en résolution virgule-fixe, que celle illustrée à la figure 3.4. Cette image fut obtenue à l'aide d'une implémentation du filtre de Wiener exécutée en résolution virgule-fixe, provenant de la plateforme de validation d'algorithmes de traitement vidéo utilisant la nouvelle OIQM combinant les métriques globale et locale. Les images, produites par la nouvelle implémentation du filtre, ne contiennent pas d'anomalie visuelle, à l'opposé de l'image de la figure 3.4. Dans le tableau 3.2, les résultats, obtenus à l'aide de la première solution, ainsi qu'à l'aide de la nouvelle OIQM, sont rapportés pour neuf images différentes [47]. Pour le cas de la première solution, l'objectif de performance était de 0,9832, alors que pour le cas de la nouvelle OIQM, l'objectif de performance était de 0,95. Finalement, l'objectif du seuil PT maximal était de 34. Chaque image originale contient du bruit gaussien avec un niveau de variance de 0,01. Les résultats obtenus démontrent que la métrique globale OIQM utilisée, soit la métrique MSSIM dans le cas présent, fournit des scores finaux très similaires.

(a) (b) (c)

Figure 3.11 Résultats de simulation obtenus avec la nouvelle OIQM. (a) Image filtrée *Cameraman* en virgule-flottante, MSSIM = 0,739830; (b) Image filtrée *Cameraman* en virgule-fixe, MSSIM = 0,729231; (c) Carte des indices SSIM comparant l'image *Cameraman* filtrée en virgule-flottante et virgule-fixe.

Par contre, les différences maximales observées entre les pixels (DMOP) sont très différentes. Il est à spécifier que les résultats obtenus avec la nouvelle OIQM ne contiennent pas d'anomalies visuelles (voir les figures 3.10, 3.11, et 3.12) alors que la première solution en produit toujours (voir les figures 3.6, 3.7, et 3.8).

Tableau 3.2 Comparaison entre la première solution et la nouvelle OIQM.

Image	**Première solution**		**Nouvelle OIQM**	
	MSSIM	DMOP	MSSIM	DMOP
Arctichare	0,702804	48	0,711137	33
Boat	0,776457	51	0,795193	32
Cameraman	0,711923	81	0,729231	31
Cat	0,807311	72	0,832029	33
Couple	0,745483	56	0,773001	27
Fruits	0,731802	69	0,745399	28
Goldhill	0,746624	67	0,754731	28
Lena	0,775641	52	0,788909	26
Peppers	0,835310	43	0,849463	25

(a) (b) (c)

Figure 3.12 Résultats de simulation obtenus avec la nouvelle OIQM. (a) Image filtrée *Lena* en virgule-flottante, MSSIM = 0,811397; (b) Image filtrée *Lena* en virgule-fixe, MSSIM = 0,788909; (c) Carte des indices SSIM comparant l'image *Lena* filtrée en virgule-flottante et virgule-fixe.

3.2 Validation de paramètres

Un grand nombre d'algorithmes de traitement vidéo sont contrôlés par des paramètres devant être fixés afin de pouvoir les utiliser. Un bon ajustement de ces paramètres peut avoir un impact significatif sur la performance et le comportement de l'algorithme. Ces paramètres sont habituellement déterminés empiriquement par les concepteurs de matériel [37] [29] [22]. Il y a donc ici le besoin d'une méthodologie afin d'automatiser la détermination et la validation des paramètres contrôlant les algorithmes de traitement vidéo.

La méthodologie de validation d'algorithmes de traitement vidéo fut donc modifiée afin de rencontrer deux objectifs principaux. Le premier objectif consiste à déterminer l'ensemble des paramètres de contrôle d'un algorithme de traitement vidéo maximisant la qualité des images produites par cet algorithme. Trouver cet ensemble de paramètres est un processus d'optimisation complexe. Afin de bien saisir la complexité de ce processus d'optimisation, prenons à titre d'exemple le

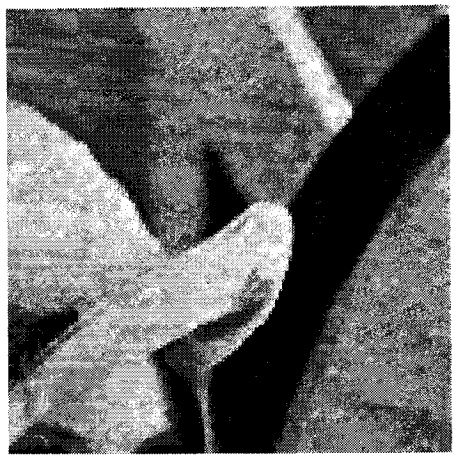


Figure 3.13 Section de l'image *Lena* filtrée en résolution virgule-fixe et ne contenant aucune anomalie.

filtre Hybride décrit dans la section 2.3.1.2 et ayant 2 paramètres à optimiser. Le premier paramètre, dénoté T_s, peut prendre 75 valeurs différentes et le second paramètre, dénoté σ_n^2, peut prendre 256 valeurs différentes. Donc, il y a $75 \times 256 = 19200$ possibilités pour l'ensemble des paramètres afin d'implémenter le filtre Hybride. Comme une simulation logicielle de filtre Hybride, évaluant 4 images, prend 81,3 secondes de temps de calcul sur une station *SPARC SunFire V440*, ayant 8 Go de mémoire RAM, il faudrait 18,07 jours pour simuler l'ensemble des paramètres afin de déterminer la combinaison de paramètres optimale avec cette base de données restreinte d'images.

Le second objectif consiste à déterminer la combinaison des tailles de mots afin de minimiser le coût de l'implémentation matérielle de l'algorithme de traitement vidéo selon un objectif de performance déterminé par le concepteur. Dans l'exemple du filtre Hybride, il y a 23 opérandes à optimiser, et chacune d'entre elles possède 32 possibilités de configurations différentes. Donc, il y a $32^{23} = 4,2 \times 10^{34}$ différentes façons d'implémenter le filtre Hybride. Dans ce cas, en utilisant la même station *UNIX* sous les mêmes conditions, il faudrait $1,1 \times 10^{29}$ années pour simuler toutes les possibilités afin de déterminer la combinaison optimale des tailles de mot du

chemin de données de l'algorithme. Les modifications apportée à l'outil AWLDT, afin que la méthodologie puisse automatiquement estimer les paramètres d'un algorithme de traitement vidéo tout en optimisant et validant son implémentation matérielle, sont décrites dans la section 3.2.1. Par la suite, la section 3.2.3 présente les résultats obtenus à l'aide de la nouvelle plateforme de validation d'algorithmes de traitement vidéo.

3.2.1 Modification de la méthodologie

Le problème concernant l'optimisation de fonctions contenant plusieurs paramètres et selon de multiples objectifs est bien connu dans la littérature et plusieurs méthodes sont proposées afin d'adresser ce dernier [2] [31]. À partir des algorithmes de traitement vidéo analysés pour cette section, il fut observé à l'aide de multiples simulations sur différentes images et séquences vidéo que le domaine des solutions d'intérêt des paramètres de chaque algorithme ne contenait pas de multiples minimum locaux. Donc, une approche simple, basée sur une descente de gradient, fut sélectionnée afin de déterminer l'ensemble des valeurs optimales de chacun des paramètres. Il est clair que la méthode proposée pourrait bénéficier grandement de méthodes d'optimisation plus complexes comme les algorithmes génétiques ou le recuit simulé si, et seulement si, il existe des minimums locaux à des endroits non prévisibles. Dans le cas contraire, leur utilisation ralentirait grandement la méthodologie proposée, car ces algorithmes sont très coûteux en termes de temps de calcul logiciel.

Afin de guider l'outil AWLDT vers la solution optimale, deux indices de qualité sont calculés. Le premier indice de qualité, dénoté Q_1, est calculé comme suit:

$$Q_1 = \frac{1}{M} \sum_{m=0}^{M-1} \frac{MSSIM(I_m^o, I_m^n)}{MSSIM(I_m^o, I_m^f)} \tag{3.2}$$

où M représente le nombre d'images à traiter, I_m^o représente la m^{ieme} image originale (I^o), I_m^n représente la m^{ieme} image d'entrée bruitée affectée par un type de bruit défini (I^n), I_m^f représente la m^{ieme} image de sortie filtrée (I^f) par l'algorithme de traitement vidéo, et $MSSIM(\cdot)$ représente la métrique objective de mesure de la qualité d'images décrite dans la section 2.1.2.1. Le deuxième indice de qualité, dénoté Q_2, est calculé comme suit:

$$Q_2 = \min_{m=0,M-1} \left\{ \frac{MSSIM(I_m^o, I_m^n)}{MSSIM(I_m^o, I_m^f)} \right\} \tag{3.3}$$

où $min(\cdot)$ représente la fonction minimum.

Le but de la première phase est de trouver la solution maximisant la somme des indices de qualité en utilisant un poids égal pour chacun d'entre eux. Pour chacun des paramètres P_l, $l = 0, 1, \ldots, L-1$ où P_l représente la valeur d'un paramètre considéré et L représente le nombre de paramètres à estimer, le concepteur de matériel doit définir les valeurs des limites supérieures P_l^{max} et inférieures P_l^{min}. De plus, le concepteur doit définir le pas de résolution minimum S_l^{min} pour chacun des paramètres.

La seconde phase est effectuée pratiquement de la même façon que celle décrite dans la section 3.1.1.2, avec pour particularité que les paramètres sont traités en tant qu'opérandes. De plus, au lieu de modifier la taille de mot, comme pour chaque opérande, en additionnant ou en soustrayant un bit, l'outils AWLDT modifie la valeur de chaque paramètre en augmentant ou en réduisant ce dernier par sa valeur S_l^{min}. Cette phase utilise deux métriques de mesure de qualité d'image, soit la métrique globale IRATIO et la métrique locale PT. Le fait d'optimiser simultanément (i.e. dans un seul processus) la taille de mot de chaque opérande et la valeur de chaque paramètre permet d'ajuster précisément chaque valeur afin de réduire l'erreur due à la précision finie du chemin de données.

Cette méthodologie, combinant deux phases d'optimisation, fut implémentée et appliquée avec succès sur trois algorithmes de traitement vidéo. La section 3.2.3 présente les résultats obtenus, à l'aide de la méthodologie d'estimation de paramètres et de la validation d'implémentation matérielle d'algorithme de traitement vidéo, pour ces trois algorithmes.

3.2.2 Fichier de simulation

Comme l'outil AWLDT est basé sur la simulation de l'algorithme afin d'estimer et de trouver les tailles optimales des opérandes, ce dernier a besoin d'un fichier de simulation. Ce fichier est généré par la plateforme de validation d'algorithmes de traitement vidéo. Dans le cas d'un réducteur de bruit avec paramètres, le fichier contiendra, pour chacune des images, la taille de l'image courante, l'image courante de référence et l'image courante bruitée. Le type de bruit, ainsi que son niveau d'intensité, doit être sélectionné par le concepteur. Le fichier modèle C/C++ qui permet de simuler un algorithme réducteur de bruit avec ses paramètres dans l'outil AWLDT se trouve à l'annexe E dans la section E.2. De plus, le fichier modèle C/C++ qui permet de calculer l'objectif de performance dans l'outil AWLDT se trouve dans la même section.

Par la suite, dans le cas d'un algorithme de dé-entrelacement vidéo, le concepteur doit fournir les images entières de chacune des séquences. La plateforme de validation d'algorithmes de traitement vidéo transforme par la suite ces séquences en champs pairs et impairs, i.e. en séquences d'images entrelacées. Finalement, le fichier de simulation qui sera généré contiendra, pour chacune des séquences, la taille de l'image courante, le nombre de champs de la séquence courante, l'image de référence et chacun des champs de la séquence courante. Le fichier modèle C/C++ qui permet de simuler un algorithme de dé-entrelacement vidéo dans l'outil

AWLDT se trouve à l'annexe E de la section E.3. De plus, le fichier modèle C/C++ qui permet de calculer l'objectif de performance pour ce type d'algorithme dans l'outil AWLDT se trouve dans la même section.

3.2.3 Résultats obtenus

Afin de tester notre méthodologie, trois algorithmes de traitement vidéo, partiellement définis par des paramètres, furent sélectionnés en tant que modèles de référence. Le premier et le second algorithme, décrits dans les section 2.3.1.2 et 2.3.1.3 respectivement, sont des filtres spatiaux. Le troisième algorithme, décrit dans la section 2.3.2.1, est un algorithme de dé-entrelacement de séquences vidéo. Les résultats obtenus pour ces trois algorithmes sont rapportés dans le tableau 3.3. Les résultats pour le filtre Hybride furent obtenus à partir de quatre images, provenant de la banque d'images de [47]. De plus, l'algorithme possède deux paramètres à estimer ainsi que vingt-trois opérandes à optimiser. Les résultats pour le filtre SUSAN furent obtenus à partir de neuf images, provenant de la banque d'images de [47]. De plus, l'algorithme possède deux paramètres à estimer ainsi que sept opérandes à optimiser. Finalement, les résultats pour l'algorithme de dé-entrelacement du type intra-champ ELA modifié furent obtenus à partir de la huitième image de la séquence vidéo *Football*. Cet algorithme possède deux paramètres à estimer ainsi que quatre opérandes à optimiser.

Tableau 3.3 Résultats obtenus pour trois algorithmes de traitement vidéo différents.

Algorithmes	**Paramètres à estimer**	**Opérandes à optimiser**	**Coût matériel**	**Objectif de Performance**		**Performance Obtenue**		**Temps de calcul (s)**
				IRATIO	**PT**	**IRATIO**	**PT**	
Filtre Hybrid	2	23	140	0.03000	34	0.025248	28	505605
Filtre SUSAN	2	7	65	0.05000	34	0.003128	31	17040
ELA modifié	2	4	23	0.05000	34	0.035429	29	576

Les valeurs optimales estimées des paramètres du filtre Hybride, obtenues à l'aide

de la plateforme de validation d'algorithme de traitement vidéo, sont $T_s = 46$ et $\sigma_n^2 = 186.845776$. La figure 3.14 illustre les courbes de contour de l'indice de qualité Q_1 (voir l'Équation (3.2)) pour les paramètres du filtre Hybride. Ces courbes furent obtenues manuellement à l'aide de simulations C/C++ du filtre Hybride en représentation virgule-flottante et en fixant les limites du paramètre T_s dans l'intervalle $[15, 75]$, avec un pas de 2, et en fixant les limites du paramètre σ_n^2 dans l'intervalle $[10, 255]$, avec un pas de 5. Un total de 1550 simulations de l'algorithme furent obtenues pour chacune des images. La figure 3.14 fut calculée à l'aide du logiciel MATLAB, en interpolant les solutions possibles afin de produire des courbes de contour plus faciles à visualiser.

Afin de conclure cette section, il est possible d'observer dans le tableau 3.3 que les objectifs de performance furent atteints avec une marge confortable et que le temps de calcul, afin de valider l'implémentation d'un algorithme de traitement vidéo, est dépendant du nombre de paramètres à estimer et d'opérandes à optimiser. Deux techniques pour accélérer l'analyse des algorithmes de traitement vidéo seront présentées au chapitre 4. Il est est tout de même possible de remarquer que dans le cas du filtre hybride, une solution fut obtenue en 5,81 jours. Ce temps pour obtenir un résultat valide est beaucoup plus rapide que de simuler toutes les possibilités pour les valeurs des paramètres de l'algorithme, soit 18,07 jours, et pour la taille optimale des opérandes, soit $1,1 \times 10^{29}$ années.

3.3 Conclusion

Dans ce chapitre, une approche pour estimer les paramètres d'algorithmes de traitement vidéo et pour valider leur implémentation matérielle fut présentée. Le but de cette approche est de pousser la tâche d'estimation de paramètres optimaux pour des algorithmes de traitement vidéo et de validation de leur implémentation

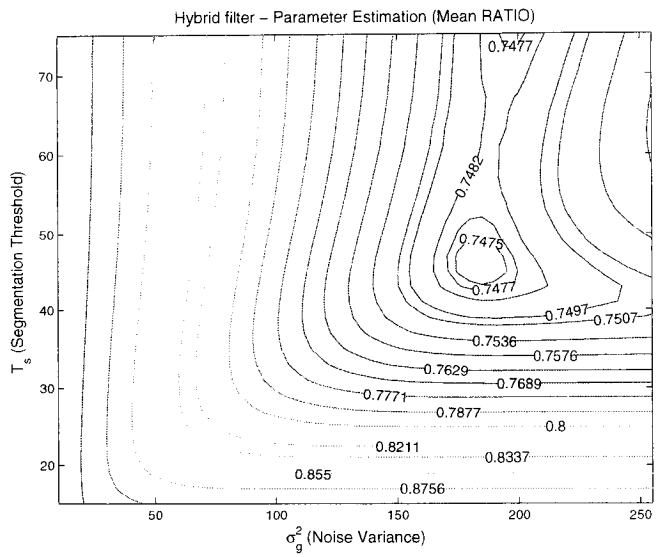


Figure 3.14 Indice de Qualité Q pour l'estimation des paramètres du filtre Hybride.

à un niveau d'abstraction plus élevé pour un concepteur matériel. Afin d'atteindre notre but, une méthodologie, prenant avantage d'un outil de détermination automatique de la longueur de mots d'un algorithme (AWLDT) et d'une nouvelle métrique objective mesurant la qualité d'image (OIQM), fut proposée.

L'utilisation d'OIQM permet de retirer un observateur humain du processus d'évaluation de la qualité d'image et guide l'outil AWLDT vers la solution optimale. Les résultats expérimentaux obtenus démontrent que la combinaison d'un objectif de performance globale et local permet d'éviter l'obtention d'implémentation d'algorithmes de traitement vidéo produisant des erreurs locales dues aux instabilités numériques. La métrique locale adoptée effectue une comparaison pixel à pixel fonctionnant bien avec les filtres spatiaux ainsi que les algorithmes de déentrelacement classique.

L'outil AWLDT fut modifié afin de déterminer automatiquement, en un seul processus d'optimisation, la combinaison des tailles de mot minimales afin de minimiser le coût de l'implémentation matérielle, tout en rencontrant la qualité d'image

spécifiée par le concepteur, en identifiant les valeurs optimales pour les paramètres de contrôle selon la précision finie du chemin de données. Cette méthodologie d'optimisation systématique et de validation automatique fut appliquée sur trois algorithmes de traitement vidéo. Les résultats expérimentaux obtenus à l'aide de ces algorithmes démontrent que l'ensemble des paramètres obtenus du processus d'optimisation concordent avec l'ensemble des valeurs optimales obtenues manuellement en simulation. Ces résultats démontrent de plus que la méthodologie proposée peut automatiquement déterminer et valider les paramètres contrôlant les algorithmes de traitement vidéo.

CHAPITRE 4

TECHNIQUES D'ACCÉLÉRATION DE LA VALIDATION D'ALGORITHMES DE TRAITEMENT VIDÉO

Dans ce chapitre, il sera question de deux techniques d'accélération de la validation d'algorithmes de traitement vidéo. Tout d'abord, la première technique concerne le regroupement d'opérandes effectuant la même opération mathématique et recevant des données de même taille. Cette technique est abordée dans [40] sur la base d'une analyse mathématique formelle, alors que l'analyse présente sera faite à l'aide de simulations. Par la suite, la deuxième technique concerne l'analyse autonome du regroupement d'opérandes en sous-ensembles faisant partie d'une structure parallèle. Ce chapitre pose comme hypothèse que l'analyse des opérandes et des paramètres d'un algorithme de traitement vidéo, à l'aide des deux techniques proposées, est moins coûteuse en terme de temps de simulation, tout en procurant sensiblement le même coût matériel. Ce chapitre est divisé en trois sections. La section 4.1 présente l'algorithme de dé-entrelacement adaptatif au mouvement servant de modéle de référence pour l'étude effectuée dans ce chapitre. Par la suite, la section 4.2 présente les résultats obtenus à l'aide des deux techniques d'accélération de l'analyse. Pour terminer, la section 4.3 formule les conclusions concernant ce chapitre et propose quelques travaux futurs.

4.1 Algorithme de dé-entrelacement adaptatif au mouvement

L'algorithme utilisé, afin de procéder à l'analyse proposée, provient d'un algorithme de dé-entrelacement adaptatif au mouvement [25]. Ce dernier est composé

de trois algorithmes distincts fonctionnant en parallèle. Le code source C/C++ de l'algorithme de dé-entrelacement adaptatif au mouvement se trouve en référence à l'annexe F. La figure 4.1 illustre le schéma bloc des trois algorithmes qui le composent. L'algorithme de dé-entrelacement adaptatif au mouvement reçoit à l'entrée quatres champs, dénotés T_i, T_j, T_k et T_l, représentant respectivement le champ qui précède le précédant, le champs précédant, le champ courant, ainsi que le champ suivant. Tout dépendamment de la nature du champ courant, c'est-à-dire un champ contenant seulement des lignes paires ou seulement des lignes impaires, l'algorithme interpole les lignes manquantes du champ courant afin de reconstituer l'image en entier. Les trois composantes de cet algorithme seront décrites aux sections 4.1.1, 4.1.2 et 4.1.3. La première composante de l'algorithme consiste à détecter le mouvement à l'intérieur du champ courant afin d'appliquer soit l'algorithme intra-champ, dans le cas d'une région dynamique, soit l'algorithme inter-champ, dans le cas d'une région statique. La deuxième composante de l'algorithme consiste en un algorithme de dé-entrelacement de type intra-champ qui est plus efficace pour les séquences vidéo contenant beaucoup de mouvement. Pour terminer, la troisième composante de l'algorithme consiste en un algorithme de dé-entrelacement de type inter-champ qui est plus efficace pour les séquences vidéo possédant peu ou pas de mouvement.

4.1.1 Algorithme de détection de mouvement

Cet algorithme a pour principale tâche de déterminer, pour chacun des pixels à interpoler, si la zone courante est une région avec du mouvement, une région avec frontière ou une région statique. Afin de déterminer l'information relative au mouvement, l'algorithme utilise les résultats de calculs de trois différences d'intensité entre des pixels provenant de quatre champs, illustré par la figure 4.2. Ces trois différences d'intensité entre les pixels des quatre champs sont définies par les

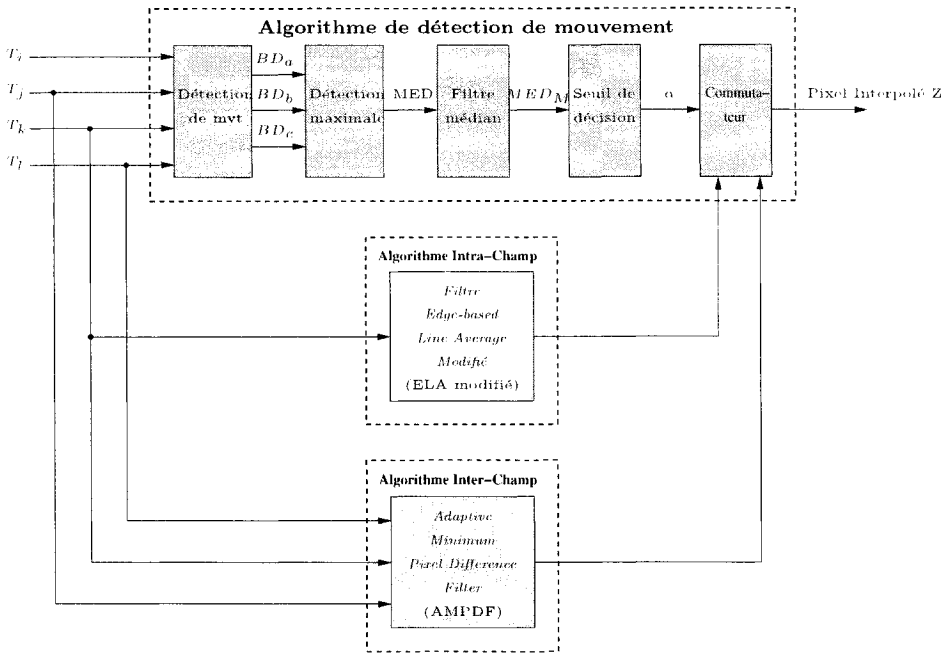


Figure 4.1 Schéma de l'algorithme de dé-entrelacement adaptatif au mouvement.

équations (4.1), (4.2) et (4.3).

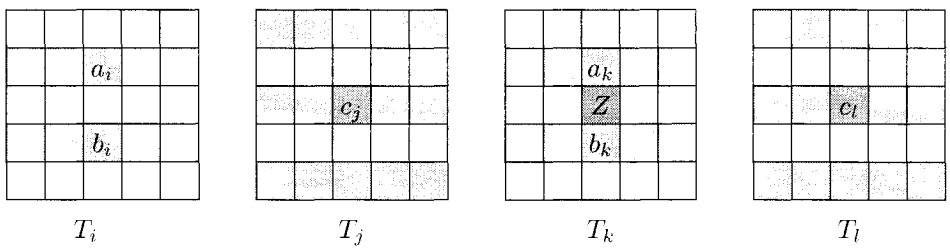


Figure 4.2 Détection de mouvement à l'aide de quatre champs de référence.

$$BD_a = |a_k - a_i|, \tag{4.1}$$

$$BD_b = |b_k - b_i|, \tag{4.2}$$

$$BD_c = |c_l - c_j|. \tag{4.3}$$

L'organigramme de l'algorithme qui effectue la classification du pixel Z à interpoler,

à l'aide des trois différences d'intensité BD_a, BD_b, et BD_c, est illustré à la figure 4.3 et son résultat est décrit par l'équation (4.4). Lorsque l'algorithme classifie la zone comme étant une région en mouvement, la variable α prend la valeur 1 et le résultat de l'algorithme ELA modifié (voir section 2.3.2.1) est appliqué au pixel à interpoler Z. Dans le cas où l'algorithme classifie la zone comme étant une région statique, la variable α prend la valeur 0 et le résultat de l'algorithme Adapdive Minimum Pixel Difference Filter (AMPDF), présenté à la section 4.1.3, est appliqué au pixel à interpoler Z. Dans le cas contraire, l'algorithme classifie la zone comme étant une région avec frontière. Les valeurs possibles que peuvent prendre la variable α sont décrites par l'équation (4.5).

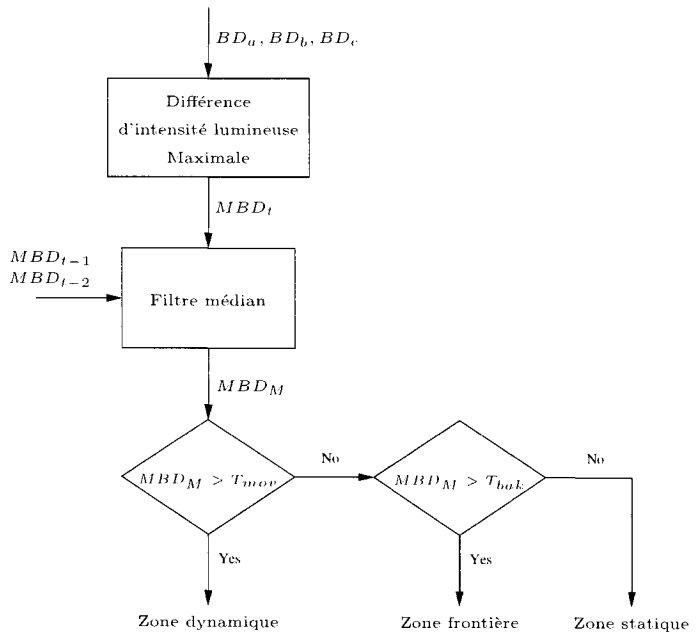


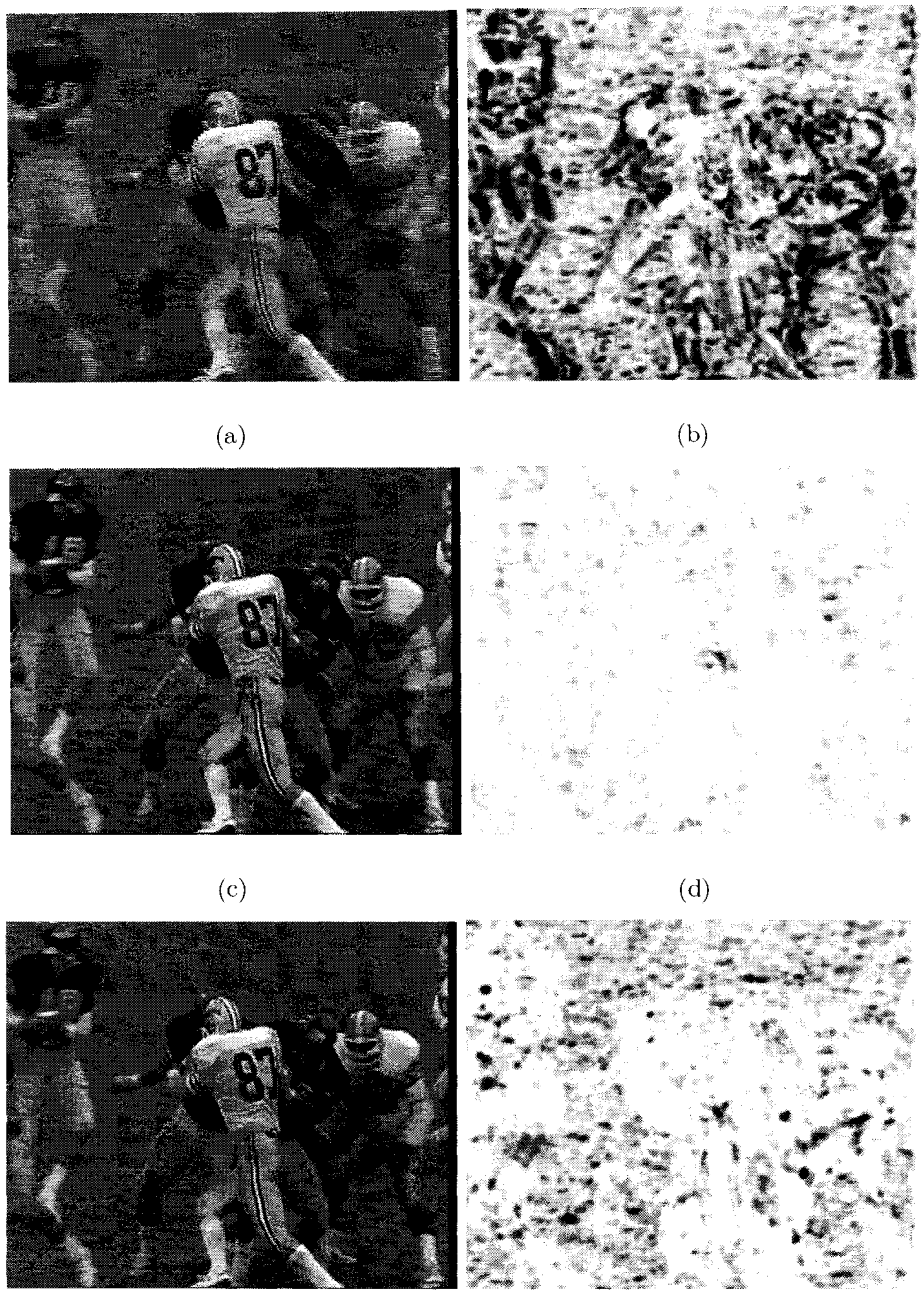
Figure 4.3 Schéma de de la classifiation de l'algorithme de détection de mouvement.

$$Z = (1-\alpha) \cdot Z_{AMPDF} + \alpha \cdot Z_{ELA}, \tag{4.4}$$

$$\alpha = \begin{cases} 1 & \text{, région avec mouvement} \\ \frac{MBD_M - T_{bak}}{T_{mov} - T_{bak}} & \text{. région avec frontière} \\ 0 & \text{, région statique} \end{cases} \quad (4.5)$$

Les figures 4.4 et 4.5 illustrent le dé-entrelacement du champ 9 des séquences *football* et *stennis* respectivement. Pour ces deux séquences, l'algorithme de détection de mouvement emploie le paramètre $T_{mov} = 34$ et le paramètre $T_{bak} = 7$, déterminés chacun par notre plateforme de validation d'algorithmes de traitement vidéo. Les figures 4.4(a) et 4.5(a) représentent les deux images entières dé-entrelacées à l'aide de l'algorithme AMPDF et les figures 4.4(b) et 4.5(b) représentent les résultats locaux de la métrique objective de qualité d'image MSSIM. Par la suite, les figures 4.4(c) et 4.5(c) représentent les images entières dé-entrelacée à l'aide de l'algorithme ELA modifié et les figures 4.4(d) et 4.5(d) représentent les images des résultats locaux de la métrique MSSIM. Finalement, les figures 4.4(e) et 4.5(e) représentent les images entières dé-entrelacée à l'aide de l'algorithme de dé-entrelacement adaptatif au mouvement, afin d'illustrer le comportement de l'algorithme de détection de mouvement, et les figures 4.4(f) et 4.5(f) représentent les images des résultats locaux de la métrique MSSIM. Dans le cas des résultats locaux de la métrique MSSIM, les valeurs varient dans l'intervalle $[0, 1]$. Afin de représenter chacun des résultats sous la forme de pixels afin d'obtenir une image entière, ceux-ci sont multipliés par la valeur 255. Les régions blanches représentent alors des valeurs de métriques élevées, tandis que les régions noires représentent des valeurs de métriques faibles.

Le tableau 4.1 fournit les résultats de la qualité des images, calculés avec la métrique de qualité vidéo MSSIM, par les différents sous algorithmes de dé-entrelacement faisant partie de l'algorithme de dé-entrelacement adaptatif au mouvement, appliqués aux séquences présentées aux figures 4.4 et 4.5. La séquence *football* con-

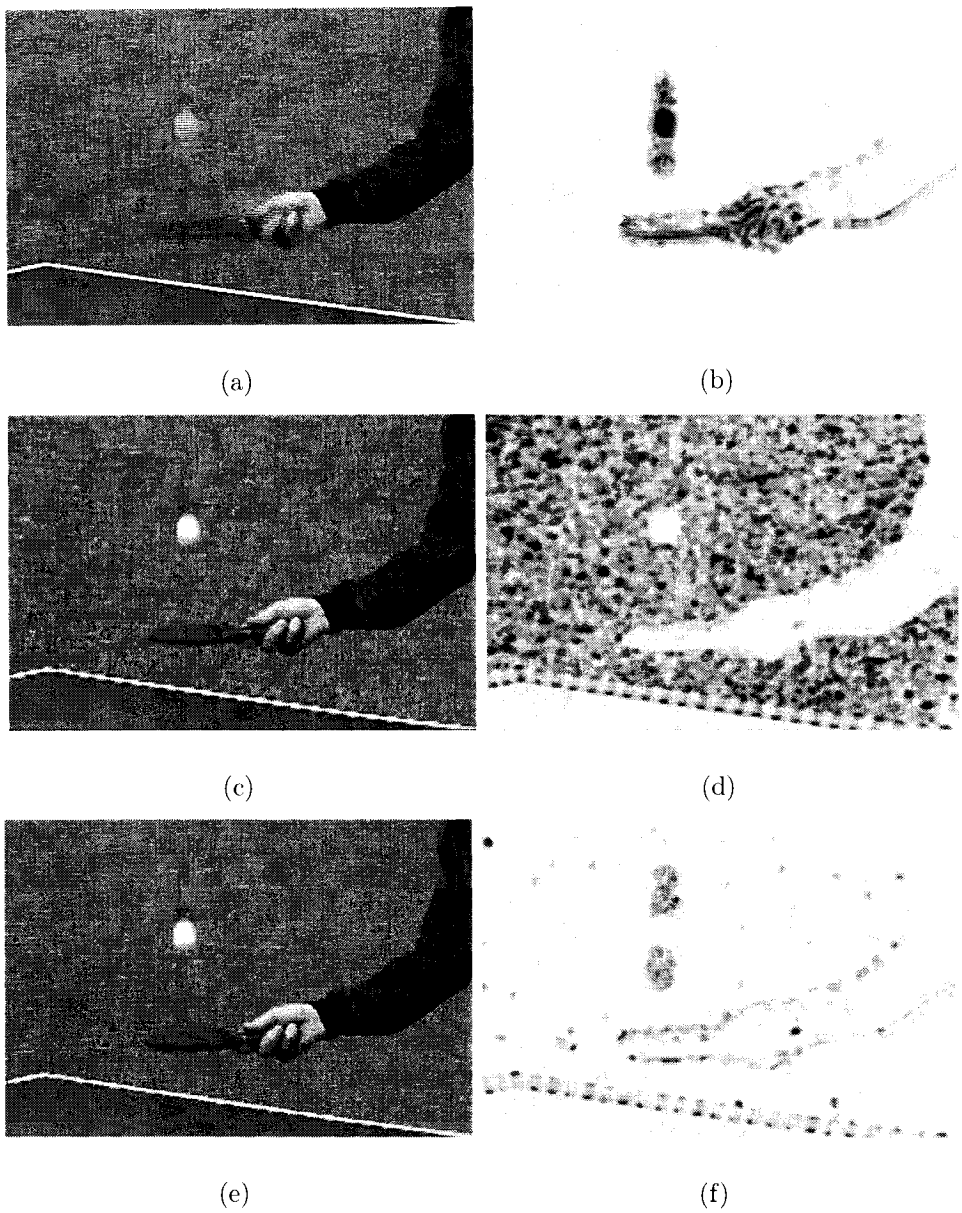


(a) (b)

(c) (d)

(e) (f)

Figure 4.4 Dé-entrelacement du champ 9 de la séquence *football*, par l'algorithme (a) ELA modifié, (b) AMPDF, (c) adaptatif au mouvement, et carte des indices SSIM pour l'algorithme (d) ELA modifié, (e) AMPDF, (f) adaptatif au mouvement.



(a) (b)

(c) (d)

(e) (f)

Figure 4.5 Dé-entrelacement du champ 9 de la séquence *stennis* par l'algorithme (a) ELA modifié, (b) AMPDF, (c) adaptatif au mouvement, et carte des indices SSIM pour l'algorithme (d) ELA modifié, (e) AMPDF, (f) adaptatif au mouvement.

tient plusieurs éléments de mouvement, alors que la séquence *stennis* contient plutôt plusieurs éléments statiques. C'est la raison pour laquelle il est logique d'observer dans le tableau 4.1 que l'algorithme ELA modifé est plus efficace pour la première séquence, alors que l'algorithme AMPDF est plus efficace pour la deuxième séquence. Les résultats de l'algorithme de détection de mouvement démontrent que l'algorithme de dé-entrelacement adaptatif au mouvement est efficace pour les deux séquences. Par contre, l'algorithme de détection de mouvement peut effectuer une mauvaise détection résultant à l'application du mauvais algorithme de dé-entrelacement pour le pixel courant. Cela explique le résultat inférieur de l'indice MSSIM pour l'algorithme de détection de mouvement lors de la séquence *footbal* par rapport à l'algorithme ELA modifié.

Tableau 4.1 Qualité des images obtenues avec différents algorithmes de dé-entrelacement.

Algorithme	**Séquence *football***	**Séquence *stennis***
AMPDF	0.647615	0.942922
ELA modifié	0.952369	0.715651
Détection de mouvement	0.860798	0.951585

4.1.2 Algorithme de dé-entrelacement ELA modifié

La description de l'algorithme de dé-entrelacement ELA modifié se trouve dans la section 2.3.2.1. Cet algorithme utilise le champ dénoté T_k afin d'interpoler les pixels des lignes manquantes. Pour les deux séquences, présentées par les figures 4.4 et 4.5, l'algorithme emploie les paramètres $T_d = 50$ et $T_v = 102$, déterminés chacun par notre plateforme de validation d'algorithmes de traitement vidéo.

4.1.3 Algorithme de dé-entrelacement AMPDF

L'algorithme AMPDF [25] est un algorithme de dé-entrelacement de type inter-champ. Afin de procéder à l'interpolation des pixels des lignes manquantes, cet algorithme utilise trois champs de références dénotés T_j, T_k et T_l, illustrés à la figure 4.6, et passe à travers trois étapes. Le schéma décrivant les différentes étapes afin d'interpoler le pixel courant Z est illustré à la figure 4.7.

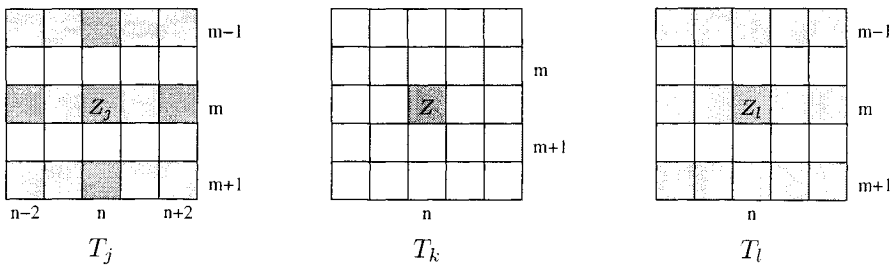


Figure 4.6 Dé-entrelacement à l'aide de trois champs de référence.

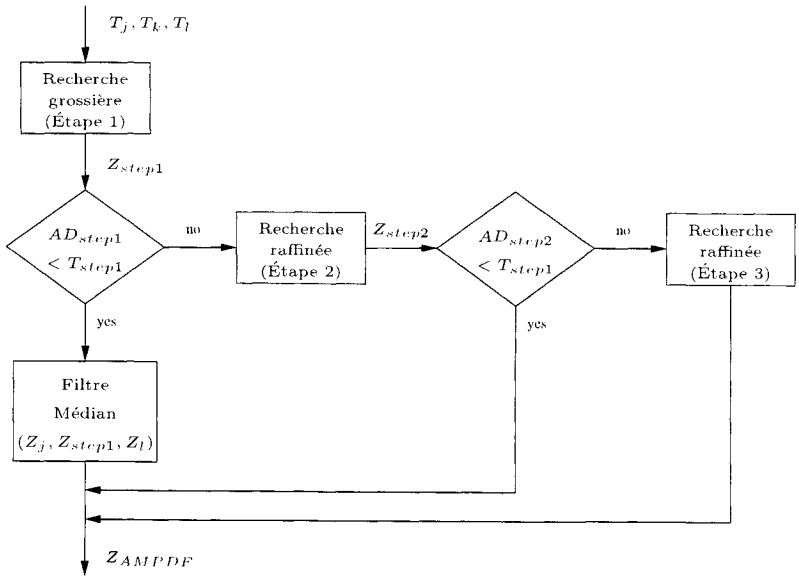


Figure 4.7 Schéma du fonctionnement de l'algorithme AMPDF.

La première étape de l'algorithme consiste à calculer la différence absolue entre 5 pixels provenant du champ T_j et le pixel de référence Z_l, provenant du champ T_l. Le

pixel ayant la plus petite différence absolue est sélectionné et est dénoté Z_{step1}. Si la différence minimale est plus petite que le seuil T_{step1}, un filtre médian est utilisé afin d'obtenir la valeur finale du pixel à interpoler. La recherche de la différence absolue minimale est décrite par l'équation (4.6).

$$AD_{step1} = min \left\{ \begin{array}{l} |Z_l(n,m) - Z_j(n,m-1)|, \\ |Z_l(n,m) - Z_j(n-2,m)|, \\ |Z_l(n,m) - Z_j(n,m)|, \\ |Z_l(n,m) - Z_j(n+2,m)|, \\ |Z_l(n,m) - Z_j(n,m+1)| \end{array} \right\} \tag{4.6}$$

Si la différence minimale de la première étape est plus grande que le seuil T_{step1}, l'algorithme passe à la deuxième étape. Celle-ci consiste au calcul de la différence absolue entre 4 pixels provenant du champs T_k et le pixel de référence Z_l. Les positions verticale et horizontale, dénotées respectivement x_1 et y_1, proviennent du pixel nommé Z_{step1}. Le pixel ayant la plus petite différence absolue est sélectionné et est dénoté Z_{step2}. La recherche de la différence absolue minimale est décrite par l'équation (4.7).

$$AD_{step2} = min \left\{ \begin{array}{l} |Z_l(n,m) - Z_k(x_1-1,y_1)|, \\ |Z_l(n,m) - Z_k(x_1-1,y_1)+1|, \\ |Z_l(n,m) - Z_k(x_1+1,y_1)|, \\ |Z_l(n,m) - Z_k(x_1+1,y_1+1)| \end{array} \right\} \tag{4.7}$$

Si la différence minimale est plus grande que le seuil T_{step2}, l'algorithme raffine sa

recherche une seconde fois et passe à l'étape 3. La troisième et dernière étape de l'algorithme consiste à calculer la différence absolue entre 4 pixels provenant des champs T_j et T_k et le pixel de référence Z_l. Les positions verticale et horizontale, dénotées respectivement x_2 et y_2, proviennent du pixel nommé Z_{step2}. Le pixel ayant la plus petite différence absolue devient alors la valeur du pixel à interpoler. La recherche de la différence absolue minimale est décrite par l'équation (4.8).

$$AD_{step3} = min \left\{ \begin{array}{l} |Z_l(n,m) - Z_j(x_2, y_2 - 1)|, \\ |Z_l(n,m) - Z_j(x_2, y_2)|, \\ |Z_l(n,m) - Z_k(x_2 - 1, y_2)|, \\ |Z_l(n,m) - Z_k(x_2 + 1, y_2)| \end{array} \right\} \tag{4.8}$$

Pour les deux séquences, présentées par les figures 4.4 et 4.5, l'algorithme emploie les paramètres $T_{step1} = 55$ et $T_{step2} = 85$, déterminés chacun par notre plateforme de validation d'algorithmes de traitement vidéo.

4.2 Résultats obtenus

Les résultats, présentés aux sections 4.2.1 et 4.2.2, ont été obtenus à l'aide de deux séquences vidéo, soit les champs 7, 8, 9, et 10 de la séquence *football*, ainsi que les champs 7, 8, 9, et 10 de la séquence *stennis*. Dans les deux cas, les champs 7 et 9 sont de type paire, et les champs 8 et 10 sont de type impaire. Donc, l'algorithme de dé-entrelacement doit reconstituer l'image 9 en entier, i.e. interpoler les lignes impaires de l'image. Les implémentations de l'algorithme de dé-entrelacement adaptatif au mouvement en résolution virgule-fixe proviennent de notre plateforme de validation d'algorithmes de traitement vidéo avec comme objectifs de performance

IRATIO fixé à 0.05 et PT fixé à 34.

La première technique d'accélération consiste à regrouper des opérandes effectuant la même opération mathématique et recevant des données de même taille. La figure 4.8 illustre un exemple du regroupement d'opérandes. À l'intérieur du DFG ordonnancé de cette figure, les quatres opérations d'addition se trouvant dans le rectangle en lignes pointillées rouges qui peuvent être regroupées en une seule opérande à analyser. Par la suite, la deuxième technique d'accélération consiste à analyser de façon autonome le regroupement d'opérandes en sous-ensemble faisant partie d'une structure parallèle. Par exemple, l'algorithme de dé-entrelacement adaptatif au mouvement, illustré à la figure 4.1, peut être décomposé en trois sous-ensembles.

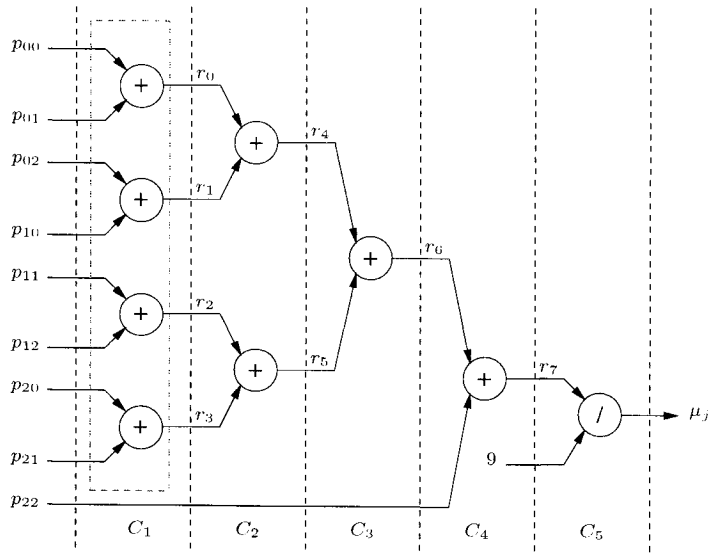
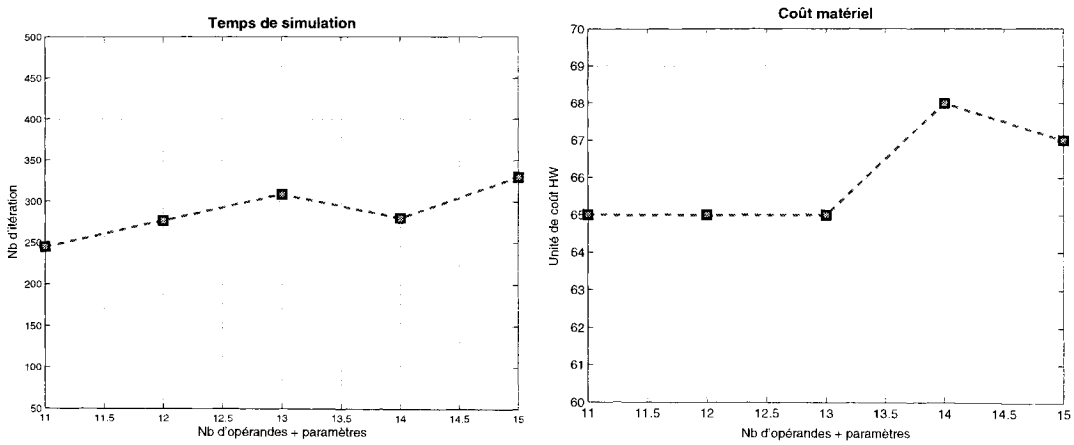


Figure 4.8 DFG ordonnancé illustrant un exemple du regroupement d'opérandes.

4.2.1 Première technique d'accélération de l'analyse

Pour la première technique d'accélération, l'algorithme de détection de mouvement, décrit à la section 4.1.1, fut utilisé en tant que modèle de référence. Lors de la

première étape de l'algorithme, trois différences d'intensités sont effectuées (voir équations (4.1), (4.2) et (4.3)). Celles-ci peuvent être divisées en 2 opérations, soit une soustraction et une valeur absolue. Les résultats obtenus à l'aide de cet algorithme sont illustrés à la figure 4.9.

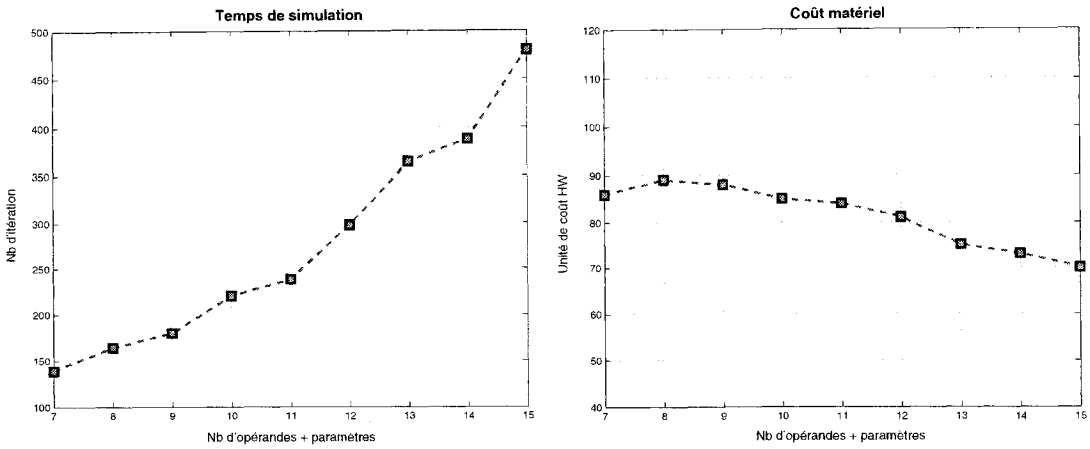


(a) (b)

Figure 4.9 Résultats de simulation pour l'algorithme de détection de mouvement et variation des (a) temps de simulation et (b) coût matériel en fonction du nombre d'opérandes et de paramètres.

Un deuxième modèle de référence fut utilisé, soit l'algorithme ELA modifié. La première étape de cet algorithme consiste à effectuer cinq différences directionnelles (voir équations (2.20), (2.21), (2.22), (2.23) et (2.24)). Comme pour l'algorithme de détection de mouvement, celles-ci peuvent être divisées en 2 opérations, soit une soustraction et une valeur absolue. Les résultats obtenus à l'aide de cet algorithme sont illustrés à la figure 4.10.

Pour les deux algorithmes, on peut clairement observer, à l'aide des courbes des graphiques 4.9(a) et 4.10(a), que le temps de simulation augmente avec le nombre de paramètres et opérandes à analyser. Par contre, pour ce qui est du coût matériel, mesuré en nombre de bits de largeur totaux dans l'ensemble des



(a) (b)

Figure 4.10 Résultats de simulation pour l'algorithme ELA modifié et variation des (a) temps de simulation et (b) coût matériel en fonction du nombre d'opérandes et de paramètres.

opérandes, les deux algorithmes n'ont pas les mêmes tendances. Dans le cas de l'algorithme de détection de mouvement, la courbe du graphique 4.9(b) démontre que le coût matériel augmente sensiblement avec un nombre croissant de paramètres et d'opérandes à analyser. Cependant, dans le cas de l'algorithme ELA modifié, la courbe du graphique 4.10(b) démontre que le coût matériel décroît avec le nombre de paramètres et d'opérandes à analyser. Cette tendance peut être expliquée par le fait que la quantification d'opérandes à l'intérieur d'un chemin de données peut annuler l'erreur engendrée par d'autres opérandes précédant ces dernières. Il est à noter que cette tendance fut déjà observée dans [5].

4.2.2 Deuxième technique d'accélération de l'analyse

Pour la deuxième technique d'acceleration, toutes les combinaisons d'analyse, à partir des trois sous-algorithmes composants l'algorithme de dé-entrelacement adap-

tatif au mouvement, furent effectuées, i.e. les sept combinaisons possibles décrites au tableau 4.2. Par exemple, pour la combinaison 4 de ce tableau, les algorithmes ELA modifié et AMPDF sont analysés ensemble alors que l'algorithme de détection de mouvement est analysé séparément. Lorsque les algorithmes ELA modifié et AMPDF sont analysés, l'algorithme de détection de mouvement demeure en résolution virgule-flottante. Par la suite, lorsque l'algorithme de détection de mouvement est analysé, les algorithmes ELA modifié et AMPDF demeurent en résolution virgule-flottante. Les tailles des opérandes et des paramètres des trois sous-algorithmes et de l'algorithme en entier sont données au tableau 4.3.

Tableau 4.2 Différentes combinaisons d'analyse de l'algorithme.

Combinaisons	**Détection de mouvement**	**ELA Modifié**	**AMPDF**	**Opérandes et paramètres**
1		X		7
2			X	8
3	X			11
4		X	X	15
5	X	X		18
6	X		X	19
7	X	X	X	26

Tableau 4.3 Tailles des opérandes et paramètres des algorithmes.

Algorithme de dé-entrelacement	**Opérandes**	**Paramètres**
Détection de mouvement	9	2
ELA modifié	5	2
AMPDF	6	2
Adaptatif au mouvement	20	6

La figure 4.11 illustre les résultats obtenus pour cette deuxième technique d'accélération de la validation d'algorithme de traitement vidéo. Les résultats obtenus pour le temps de simulation, illustrés par le graphique 4.11(a), démontrent que le temps de simulation augmente linéairement lorsque le nombre d'opérandes et de paramètres à analyser devient plus grand. Sur ce graphique, le temps de simulation

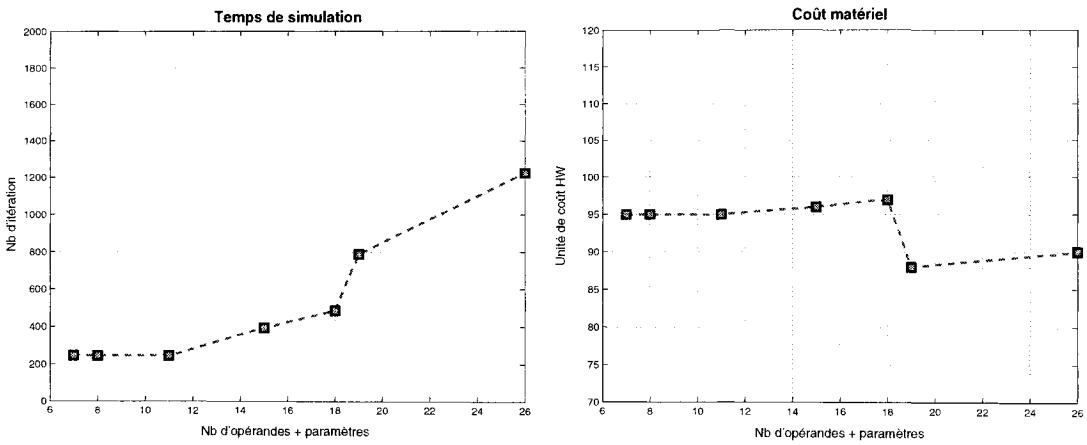
est exprimé en terme du nombre d'itérations que prend l'outil AWLDT. Chaque itération prend un temps d'exécution de 11,52 secondes lorsque l'outil est exécuté sur un *Sun Fire V890 / 16 processeurs* ayant *32 Go* de mémoire et roulant avec le système d'exploitation *Solaris 9*. La simulation de chacune des composantes de l'algorithme séparément, i.e. l'algorithme ELA modifié prenant 103 itérations, l'algorithme AMPDF prenant 127 itérations et l'algorithme de détection de mouvement prenant 245 itérations, pour un total de 475 itérations, prennent moins de temps de simulation que l'analyse de l'algorithme de dé-entrelacemnt adaptatif au mouvement en entier, car ce dernier prend 1224 itérations.

La courbe du graphique 4.11(b) démontre que le coût matériel reste sensiblement le même, soit entre 88 et 97. Il y a une légère tendance vers la hausse du coût matériel avec plus d'opérandes et de paramètres à analyser. Cette tendance est normale car lorsque certaines opérandes ne sont pas analysées, elles demeurent en résolution virgule-flottante, i.e. elles gardent leur pleine résolution, ce qui entraîne que les tailles des opérandes analysées peuvent être davantage réduites. Ceci a une répercussion sur la performance de l'algorithme. En effet, lorsque chacun des sous-algorithmes est analysé séparément, les objectifs de performance sont atteints. Cependant, lorsque les implémentations en résolution virgule-fixe de ces sous-algorithmes sont combinées ensemble pour former l'algorithme en entier, il y a une baisse de performance qui peut être observée sur la courbe du graphique 4.11(c). Cette baisse de performance de l'algorithme provenant de la combinaison des analyses vient confirmer l'hypothèse de départ, à savoir que l'analyse des opérandes et des paramètres d'un algorithme de traitement vidéo, à l'aide des deux techniques proposées, est moins coûteuse en terme de temps de simulation, tout en procurant sensiblement le même coût matériel. Pour terminer, il est normal de trouver, sur les courbes des graphiques de la figure 4.11, certains points ne suivant pas la tendance des autres données. En effet, l'outil AWLDT utilise une heuristique afin

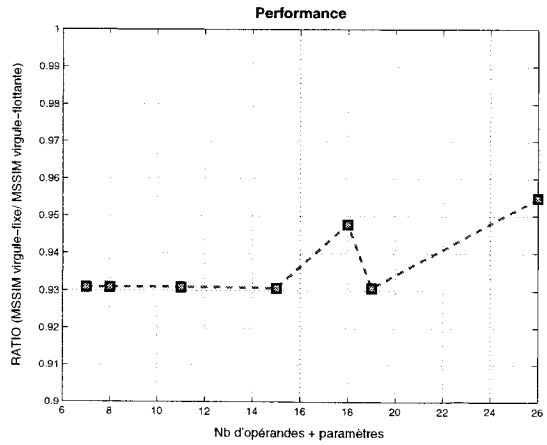
d'optimiser les paramètres ainsi que la taille des opérandes à analyser à l'intérieur de l'algorithme. Donc, l'outil AWLDT, en tentant de minimiser le coût matériel, tout en respectant les objectifs de performance, fournit non pas la solution optimale de l'implémentation en résolution virgule-fixe, mais bien une des solutions localement optimales.

4.3 Conclusion

Dans ce chapitre, deux techniques afin d'accélérer l'analyse pour la validation d'algorithmes de traitement vidéo ont été présentées. La première technique d'accélération de l'analyse porte sur le regroupement d'opérandes effectuant la même opération mathématique et recevant des données de même taille. Par la suite, la deuxième technique d'accélération de l'analyse porte sur le regroupement d'opérandes en sous-ensemble provenant d'une structure en parallèle faisant partie du même ensemble. Les résultats obtenus à l'aide de l'algorithme de dé-entrelacement adaptatif au mouvement ont confirmé les hypothèses posées au départ. En effet, de manière globale, l'analyse des opérandes et paramètres de cet algorithme est moins coûteuse en terme de temps de simulation. Le coût matériel demeure sensiblement le même, mais par contre, il y a une légère baisse au niveau de la performance de l'algorithme. Évidemment, il ne s'agit que d'un exemple. Afin de généraliser ces tendances, il faudrait pousser plus loins la présente analyse à l'aide, entre autre, d'algorithmes provenant de différentes classes d'application. De plus, il serait probablement possible d'automatiser la segmentation de l'algorithme en sous-ensembles distincts à l'aide d'une méthodologie plus formelle utilisant des modèles mathématiques afin d'analyser les dépendances de données.



(a) (b)



(c)

Figure 4.11 Résultats de simulation pour l'algorithme de dé-entrelacement adaptatif au mouvement, variation du (a) temps de simulation, (b) coût matériel, et (c) de la performance, en fonction du nombre d'opérandes et paramètres.

CHAPITRE 5

CONCLUSION

À l'intérieur de ce chapitre, la section 5.1 présente une synthèse des travaux présentés dans les chapitres 2, 3, et 4. Par la suite, la section 5.2 termine ce chapitre et propose des améliorations ainsi que des possibilités pour des travaux futurs sur la validation d'algorithme de traitement vidéo.

5.1 Synthèse du travail accompli

Dans le chapitre 2, les différents concepts concernant la méthodologie d'optimisation applicable à la mise en oeuvre d'algorithmes de traitement vidéo furent énoncés. Comme cette méthodologie doit être systématique et automatique, une métrique objective mesurant la qualité d'image, soit l'indice MSSIM, fut sélectionné afin de permettre le retrait d'observateur humain de la tâche d'évaluation de la qualité d'image. Cette métrique est basée sur la mesure des similarités entre une image de référence et une image distortionnée. L'indice MSSIM est une métrique mesurant la variation de l'information structurelle perçue à l'intérieur d'une image et est une métrique FR basée sur des caractéristiques du système visuel humain. Par la suite, les différents algorithmes servant à valider la méthodologie présentée dans ce mémoire, ainsi que les types de bruit vidéo furent présentés. Trois filtres numériques de la catégorie des filtres spatiaux furent décrits, soit le filtre Hybride, le filtre SUSAN, ainsi que le filtre adapatatif de Wiener. Ce chapitre a aussi couvert les méthodes de dé-entrelacement vidéo ainsi que l'algorithme de type intra-champ ELA modifié.

Le chapitre 3 a présenté la plateforme de validation d'algorithme de traitement vidéo développée dans le cadre des travaux de ce mémoire. Cette plateforme supporte une méthodologie qui permet d'estimer les paramètres d'algorithmes de traitement vidéo et de valider leur implémentation matérielle. Cette méthodologie, proposée dans ce chapitre, prend avantage d'un outil de détermination automatique de la longueur de mot d'un algorithme (AWLDT) et d'une nouvelle métrique objective mesurant la qualité d'image.

Cette nouvelle métrique fut développée à l'intérieur de ce chapitre. Il a été démontré que l'utilisation d'OIQM permet de retirer un observateur humain du processus d'évaluation d'image. De plus, cet OIQM guide l'outil AWLDT vers la solution optimale de l'implémentation matérielle de l'algorithme de traitement vidéo produisant des images valides. Les résultats expérimentaux obtenus dans ce chapitre démontrent que la combinaison d'objectifs global et local permet d'éviter l'obtention d'implémentation matérielle produisant des erreurs locales dues aux instabilités numériques de l'algorithme exécuté en virgule-fixe.

Par la suite, les modifications apportées à l'outil AWLDT, afin de déterminer automatiquement, en un seul processus d'optimisation, la combinaison de la taille de mot minimale afin de minimiser le coût de l'implémentation matérielle, ainsi que les valeurs optimales pour les paramètres de contrôle, furent présentées dans ce chapitre. Les résultats expérimentaux présentés à l'aide de trois algorithmes de traitement vidéo, soit le filtre Hybride, le filtre SUSAN, et l'algorithme de déentrelacement vidéo ELA modifié, ont démontré que l'ensemble des paramètres obtenus de la plateforme de validation d'algorithme de traitement vidéo concordent avec les valeurs optimales théoriques.

Finalement, dans le chapitre 4, deux techniques afin d'accélérer l'analyse de la validation d'algorithmes de traitement vidéo furent présentées. La première tech-

nique d'accélération de l'analyse présentée porte sur le regroupement d'opérandes effectuant la même opération mathématique et recevant des données de même taille. La deuxième technique d'accélération de l'analyse présentée porte sur le regroupement d'opérandes en sous-ensembles provenant d'une structure en parallèle faisant partie du même ensemble. Les résultats obtenus à l'aide de l'algorithme de dé-entrelacement adaptatif au mouvement ont confirmé les hypothèses posées au départ. En effet, de manière globale, ces techniques font en sorte que l'analyse des opérandes et paramètres de cet algorithme devient moins coûteuse en terme de temps de simulation. De plus, le coût matériel demeure sensiblement le même, mais par contre, une légère baisse au niveau de la qualité des images produites par l'algorithme fut observée.

5.2 Travaux futurs et améliorations

Ce mémoire utilise la combinaison d'une métrique globale, ainsi que d'une métrique locale afin de guider l'outil AWLDT vers des solutions produisant des images valides, soit des implémentations sans instabilité numérique. Cette métrique globale est une OIQM nommée indice MSSIM. Cette métrique fut choisie, car au moment où cette recherche a été effectuée, elle semblait être la meilleure disponible dans la littérature. Une nouvelle métrique mesurant la qualité d'image et la qualité vidéo est maintenant disponible [35]. Cette OIQM est nommée Visual Information Fidelity (VIF). C'est une métrique du type FR basée sur des caratéristiques du système visuel humain. Cette métrique se distingue des autres OIQM traditionnelles par le fait qu'en améliorant le contraste d'une image de façon linéaire sans ajouter de bruit, la métrique VIF retourne un résultat plus élevé que le résultat d'une image sans bruit. Cela signifie donc que l'image résultante peut être plus attrayante en regard de critère subjectif vis-à-vis de l'image de référence. Bref, la

métrique VIF est capable de capturer l'amélioration de la qualité visuelle dans une image. Il serait donc intéressant de remplacer la métrique globale MSSIM par la métrique VIF afin d'étudier si les résultats produits par la plateforme de validation d'algorithme de traitement vidéo seraient supérieurs à ceux obtenus dans ce Mémoire.

Pour terminer, une approche générique fut utilisée dans ce mémoire pour ce qui est de l'utilisation des métriques de qualité d'image. En effet, la même métrique fut utilisée pour différents types de bruit, ainsi que pour deux classes d'applications d'algorithme de traitement vidéo. Une étude sur les OIQM pourrait être effectuée pour des cas plus spécifiques, i.e. faire l'utilisation de métriques spécialisées. Par exemple, une métrique du type NR, se trouvant dans la littérature, est spécialisée dans la mesure d'artéfacts de compression dans une image ou dans une séquence vidéo, plus précisément elle se spécialise dans la mesure de l'effet de bloc [32]. Donc, il serait très intéressant d'utiliser une telle métrique à l'intérieur de la plateforme de validation d'algorithme de traitement vidéo afin de valider l'implémentation d'un réducteur de bruit se spécialisant dans le retrait de l'effet de bloc dans une image. Cela permettrait d'observer si l'utilisation d'une métrique spécialisée fournit une meilleure implémentation matérielle que l'utilisation d'une métrique plus générale.

BIBLIOGRAPHIE

[1] AGOSTINI, L., SILVA, I., AND BAMPI, S. Pipelined fast 2D DCT Architecture for JPEG Image Compression. *IEEE International Conference on Application-Specific Systems, Architectures, and Processors* (2003), 378–388.

[2] BELEGUNDU, A. *Optimization Concepts and Applications in Engineering.* Prentice Hall, 1999.

[3] BORRIONE, D. *La simulation et les méthodes de vérification formelle.* Lavoisier, 2002, ch. 5, pp. 139–174.

[4] BOX, G., AND MULLER, M. A note on the generation of normal random deviates. *Annals of Mathematical Statistics vol. 29* (1958), 610–611.

[5] CANTIN, M.-A. *Méthode de Détermination Automatique de la Taille des Chemins de Données.* PhD thesis, École Polytechnique de Montréal, Départment de Génie Électrique, 2005.

[6] CANTIN, M.-A., REGIMBAL, S., CATUDAL, S., AND SAVARIA, Y. An Unified Environment to Assess Image Quality in Video Processing. *Journal of Circuits, Systems and Computers vol. 13, no. 6* (December 2004), 1289–1306.

[7] CANTIN, M.-A., SAVARIA, Y., AND LAVOIE, P. A Comparison of Automatic Word Length Optimization Procedures. *IEEE International Symposium on Circuits and Systems vol. 2* (2002), 612–615.

[8] CANTIN, M.-A., SAVARIA, Y., PRODANOS, D., AND LAVOIE, P. An Automatic Word Length Determination Method. *IEEE International Symposium on Circuits and Systems vol. 5* (2001), 53–56.

[9] CARNEC, M., LE CALLET, P., AND BARBA, D. Full Reference and Reduced Reference Metrics for Image Quality Assessment. *IEEE Symposium on Signal Processing and Its Applications vol. 1* (July 2003), 477–480.

[10] CATUDAL, S., CANTIN, M.-A., AND SAVARIA, Y. Performance Driven Validation Applied to Video Processing. *WSEAS Transaction on Electronics vol. 1, no. 3* (July 2004), 568–575.

[11] CATUDAL, S., CANTIN, M.-A., AND SAVARIA, Y. Parameter Estimation Applied to Automatic Video Processing Algorithms Validation. *IEEE International Symposium on Circuits and Systems* (May 2005), 3439–3442.

[12] CESARIO, W., AND JERRAYA, A.-A. *La conception comportementale.* Lavoisier, 2002, ch. 3, pp. 65–108.

[13] CLARCKE, E., GRUMBERG, O., AND PELED, D. *Model Checking.* The MIT Press, January 2002.

[14] CUPAK, M., CATTHOOR, F., AND DE MAN, H. Efficient System-Level Functional Verification Methodology for Multimedia Applications. *IEEE Design and Test of Computers vol. 20* (March-April 2003), 56–64.

[15] DALY, S. *The visible difference predictor: An algorithm for the assessment of image fidelity.* The MIT Press, 1993, pp. 179–206.

[16] FURTH, B., AND MARQURE, O. *The Handbook of Video Databases: Design and Applications.* CRC Press, September 2003.

[17] GRANGER, E., CATUDAL, S., GROU, R., MBAYE, M. M., AND SAVARIA, Y. On Current Strategies for Hardware Acceleration of Digital Image Restoration Filters. *WSEAS Transaction on Electronics vol. 1, no. 3* (July 2004), 551–557.

[18] HAAN, G. D., AND BELLERS, E. De-Interlacing - An Overview. *Procedings of the IEEE vol. 86, no. 9* (September 1998), 1839–1857.

[19] JACK, K. *Video Demystified: A Handbook for the Digital Engineer.* Elsevier, 1996.

[20] KRALJIC, I., VERDIER, F. S., QUENOT, G., AND ZAVIDOVIQUE. B. Investigating Real-Time Validation of Real-Time Image Processing. *IEEE International Workshop on Computer Architecture for Machine Perception vol. 20* (October 1997), 116–125.

[21] LANE, T. The Independant JPEG Group's Software, Release 5b, 1990-1995. Archive site: *ftp.uu.net/graphics/jpeg/*.

[22] LEE, H., PARK, J., BAE, T., CHOI, S., AND HA, Y. Adaptive Scan Rate Up-Conversion System Based on Human Visual Characteristics. *IEEE Transactions on Consumer Electronics vol. 46, no. 4* (2000), 999–1006.

[23] LEE, J. Digital Image Enhancement and Noise Filtering by use of Local Statistics. *IEEE Transactions on Pattern Analysis and Machine Intelligence vol. 2* (1980), 165–168.

[24] LEE, J. *Digital Image Smoothing and the Sigma Filter*, vol. vol. 24. Computer Vision, Graphics and Image Processing, 1983.

[25] LEE, S.-G., AND LEE, D.-H. A Motion-Adaptive De-Interlacing Method Using an Efficient Spatial and Temporal Interpolation. *IEEE Transactions on Consumer Electronics vol. 49, no. 4* (November 2003), 1266–1271.

[26] LIM, J. *Two-Dimensional Signal and Image Processing.* Prentice Hall, 1990.

[27] LIN, S.-F., CHANG, Y.-L., AND CHEN, L.-G. Motion Adaptive Interpolation with Horizontal Motion Detection for De-Interlacing. *IEEE Transactions on Consumer Electronics vol. 49, no. 4* (2003), 1256–1265.

[28] LIS, J., AND GAJSKI, D. Synthesis from VHDL. *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (1998), 378–381.

[29] Loiseau, L. Méthode de Conception pour la Réutilisation et Optimisation Architecturale Appliqueées à un Réducteur de Bruit Vidéo. Master's thesis, École Polytechnique de Montréal, 2001.

[30] Lubin, J. *The use of psychophysical data and models in the analysis of display system performance.* The MIT Press, 1993, pp. 163–178.

[31] Onwubiki, C. *Introduction to Engineering Design Optimization.* Prentice Hall, 2000.

[32] Pan, F., Lin, X., Rahardja, S., Lin, W., Ong, E., Yao, S., Lu, Z., and Yang, X. A Locally-Adaptive Algorithme for Measuring Blocking Artifacts in Images and Videos. *IEEE Symposium on Circuits and Systems vol. 3* (May 2004), 925–928.

[33] Poynton, C. *Digital Video and HDTV Algorithms and Interfaces.* Morgan Kaufmann Publishers, 2003.

[34] Regimbal, S. Méthode de Réutilisation et de Couverture pour la Vérification Fonctionnelle de Circuits Numériques. Master's thesis, École Polytechnique de Montréal, 2003.

[35] Sheikh, H., and Bovik, A. Image Information and Visual Quality. *IEEE Transactions on Image Processing* (2005).

[36] Skocir, P., Marusic, B., and Tasic, J. A Three-Dimensional Extension of the SUSAN Filter for Wavelet Video Coding Artifact Removal. *IEEE Mediterranean Electrotechnical Conference* (May 2002), 395–398.

[37] Smith, S., and Brady, J. SUSAN - A New Approach to Low Level Image Processing. *International Journal of Computer Vision vol. 23* (1997), 45–78.

[38] Teo, P., and Heeger, D. Perceptual Image Distortion. *IEEE International Conference on Image Processing vol. 2* (November 2004), 982–986.

[39] Tong, H., Li, M., Zhang, H.-J., He, J., and Ma. W.-Y. Learning No-Reference Quality Metric by Examples. *IEEE International Conference on Multimedia Modelling* (January 2005), 247–254.

[40] Wadekar, S., and Parker, A. Accuracy Sensitive Word-Length Selection for Algorithm Optimization. *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (October 1998), 54–61.

[41] Wallace, G. The JPEG Still Picture Compression Standard. *Communications of the ACM vol. 34, no. 4* (1991), 30–44.

[42] Wang, Z., and Bovik, A. Why is Image Quality Assessment so Difficult? *IEEE International Conference on Acoustics, Speech, and Signal Processing vol. 4* (May 2002), 3313–3316.

[43] Wang, Z., Bovik, A., Sheikh, H., and Simoncelli, E. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing vol. 13, no. 4* (April 2004), 600–612.

[44] Wang, Z., Bovik, A., and Simoncelli, E. *Structural Approaches to Image Quality Assessment.* Academic Press, June 2005, ch. 8.3.

[45] Wang, Z., Sheikh, H., and Bovik, A. *Objective Video Quality Assessment.* CRC Press, September 2003, ch. 41, pp. 1041–1078.

[46] Wang, Z., and Simoncelli, E. Reduced-Reference Image Quality Assessment Using A Wavelet-Domain Natural Image Statistic Model. *SPIE Proceeding on Human Vision and Electronic Imaging vol. 5666* (March 2005), 149–159.

[47] Weber, A. The USC-SIPI Image Database: Version 5. Tech. rep., Signal and Image Processing, Institute University of Southern California, October 1997.

ANNEXE A

PERFORMANCE DRIVEN VALIDATION APPLIED TO VIDEO PROCESSING

Serge Catudal, Marc-André Cantin, and Yvon Savaria
Electrical Engineering Department, École Polytechnique de Montréal
C.P. 6079, succursale Centre-ville, Montréal (Québec), H3C 3A7
Canada.
{catudal, cantin, savaria}@grm.polymtl.ca http://www.grm.polymtl.ca

Abstract

This paper presents a systematic method to validate implementations of video processing algorithms. This is useful when translating algorithms implementations from a floating-point to fixed-point precision. The method takes advantage of an automatic word length determination tool and an Objective Image Quality Metric (OIQM) to implement a performance driven validation. Since results obtained with existing OIQM show anomalies inside images due to numerical instabilities, a new OIQM is proposed. It combines a global and a local metric to remove anomalies inside the resulting images. Experimental results obtained with the Wiener filter are presented to illustrate the validity of our approach.

Key-Words

Objective Image Quality Metric, Validation, Video Processing, Wiener Filter.

A.1 Introduction

Design of video processing modules can be split in two complementary components that relate to control and data-flow respectively. The main goal of the control is to feed the datapath with the right data at the right time and to collect results. The main goal of the datapath is to perform data transformations required by the algorithm. Video processing architectures that implement the data transformations are usually derived from algorithms previously validated with C/C++ or *Matlab/Simulink* implementations. They are usually simulated in floating-point resolution. Since floating-point implementations are relatively expensive in terms of hardware cost, a good way to minimize or optimize hardware implementation complexity, pure performance and power consumption is to use fixed-point precision.

The translation from a floating-point to a fixed-point formulation, a step towards implementation through a Register Transfer Level (RTL) hardware description, may cause some problems. For instance, the precision of each operand should be carefully selected to preserve algorithm stability and processing accuracy. The sensitivity of target algorithms to word widths of various operands is not clear a priori. Furthermore, this translation can deteriorate the behavior of a video processing algorithm by producing visual effects annoying to a human observer. Design validation helps the designer implementing complex systems with sufficiently high degree of confidence in their correctness under all circumstances. Most previous work on video processing validation deals with formal loop verification techniques [1] and system level image processing specifications validation in their target environment by emulation [2]. But these validation techniques are not useful to confirm the absence of visual artifacts in almost correct implementations.

A method for automatic sizing of video processing algorithm implementations was developed by the authors [3]. This method exploits an Automatic Word Length Determination Tool (AWLDT) [4] [5]. However, experiments conducted with the previously reported method showed that depending on the metric used, the method can produce anomalies in the resulting images. Fig. A.1 illustrates an example of a result image containing anomalies appearing as dark pixels that are due to numerical instabilities in the algorithm implementation.

Figure A.1 *Arctichare* filtered in fixed-point resolution causing numerical instability.

This motivated our quest of a more robust method. Ideally, the method should automatically assess algorithms performance, and always produce valid images, with no need for a human observer checking the result. Key components of a performance driven validation method is to combine the AWLDT with a metric that can suppress solutions that generate distorted images.

The purpose of this paper is to propose a new performance driven validation method applicable to video processing. It is based on a new Objective Image Quality Metric (OIQM). The rest of the paper is organized as follows. In Section A.2, our video processing validation methodology is introduced. Section A.3 briefly reviews the

Wiener Filtering algorithm used as a benchmark. Section A.4 presents results obtained with our video processing validation platform using previously published OIQM. Section A.5 proposes a new OIQM that prevent numerical instabilities while translating an algorithm to a fixed-point implementation. Finally, Section A.6 formulates conclusions and proposes some future work.

A.2 Video Processing Validation

Design validation helps designing and implementing complex systems to ensure with sufficiently high degree of confidence that they are correct under all circumstances [6]. The most common technique for design validation is based on tests and simulations. However, features not explored during these simulations may lead to fatal errors [6]. Formal techniques such as model checking were developed to enable exhaustive exploration of all possible behaviors of a design. However, these formal techniques are applicable to control validation [7] and generally not applicable to datapath validation.

By contrast, the method introduced in [3] automatically optimizes datapath of video processing algorithm implementation. This method is now refined to accomplish performance driven validation of video processing algorithm implementations using a platform described in Section A.2.1.

A.2.1 Video Processing Validation Platform

Our video processing validation platform (see Fig. A.2) uses the same three general steps as model checking [6], i.e. modeling, specification and verification, even if the methodology employed is very different. In the first step, the designer produces a

high level model of the algorithm datapath in the C or SystemC languages. This model can be compiled to a Control Data Flow Graph (CDFG) representation. In the second step, the designer defines performance objectives that will become properties of the model specification. In our application, the performance objective is represented has a target ratio between the OIQM of the fixed-point filtered image over the OIQM of the floating-point filtered image. By contrast, classical validation uses logics such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL), to describe the design specification properties [6]. In the third step, the designer must validate that the performance objectives specified are met by the model. In order to achieve this last step, our video processing validation platform exploits the AWLDT as opposed to a model checker for classical validation. From performance objectives and an image database, the AWLDT generates an optimized model of the algorithm datapath that meets specified performance targets for the implementation of the video processing algorithm.

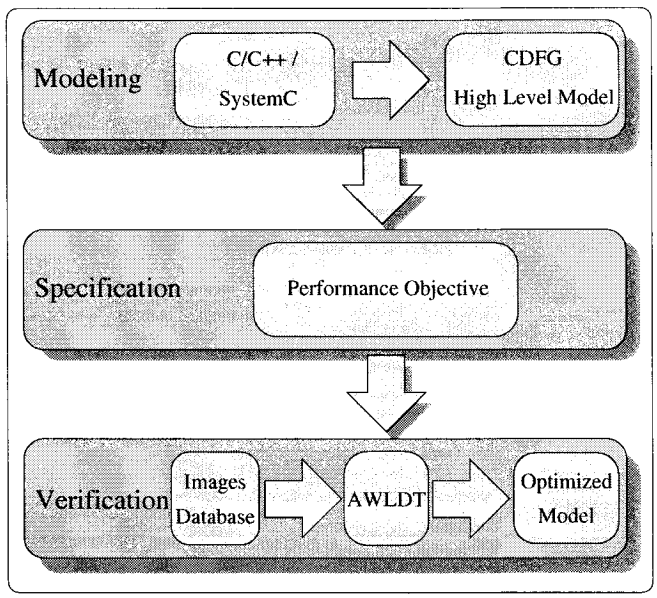


Figure A.2 Flow Diagram of the Video Processing Validation Platform

A.2.2 Objective Image Quality Metric

In our previous work [3], we used an OIQM named Universal Image Quality Index (UIQI) [8]. This metric led our optimization method to produce implementations generating unstable results when the algorithm contained steps such as division by zero. Analysis of the source of these unstable results led us to use a different OIQM that is described in Section A.2.2.1. Even with this second OIQM, we still obtained some unstable results producing visual anomalies. These anomalies are presented in Section A.4 and the new OIQM proposed to resolve this problem is described in Section A.5.

A.2.2.1 Structural Similarity Index

The second OIQM we chose for our video processing validation platform, the Structural SIMilarity (SSIM) Index [9], is based on the measurement of similarities between a reference and a distorted image. It considers image degradation as perceived structural information variation [10] [11]. The SSIM index is a full-reference image quality assessment metric based on psychophysical Human Visual System (HVS) features. By contrast, the most common OIQM employed in the literature, such as the Mean Square Error (MSE) and the Peak Signal-to-Noise Ratio (PSNR), use error sensitivity measurements by quantifying the error between the distorted image and the reference image. It has been reported that the MSE and the PSNR do not correlate well with subjective image quality measures [12].

The SSIM index combines the luminance, Eq. A.2, contrast, Eq. A.3, and structure, Eq. A.4, comparative indices. They compare a reference image x and the distorted image y to evaluate their structural similarity.

$$SSIM(x,y) = [l(x,y)]^{\alpha} \cdot [c(x,y)]^{\beta} \cdot [s(x,y)]^{\gamma}, \quad \text{(A.1)}$$

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \quad \text{(A.2)}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2\sigma_y^2 + C_2}, \quad \text{(A.3)}$$

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \quad \text{(A.4)}$$

where $C_1 = (K_1L)^2$, and $C_2 = (K_2L)^2$. Based on [9], we set in this paper $\alpha = \beta = \gamma = 1$, $C_3 = C_2/2$, $K_1 = 0.01$, and $K_2 = 0.03$. L represents the dynamic range of pixels value, and since wue use and 8-bit/pixel monotonic signal, $L = 255$.

The SSIM index is computed locally for each pixel with an 11x11 circular symmetric Gaussian weighting function. The SSIM indices computed for each position are combined to produce an SSIM index map. The mean of this SSIM index map is used to evaluate the overall image quality:

$$MSSIM(X,Y) = \frac{1}{M}\sum_{j=1}^{M} SSIM(x,y). \quad \text{(A.5)}$$

A.3 Wiener Filter

In order to test our video processing validation methodology, the adaptive Wiener filter [13], also known as the *Minimum Mean Square Error* (MMSE) algorithm was selected as benchmark. When $\nu^2 < \sigma^2$ and $\sigma^2 \neq 0$, the filtered output image F of the Wiener filter is computed as follows:

$$F(k,l) = \mu + \frac{\sigma_j^2 - \nu^2}{\sigma_j^2}(G(k,l) - \mu_j), \tag{A.6}$$

where ν^2 is the noise variance, μ and σ^2 are respectively the local mean and variance around a pixel j:

$$\nu^2 = \frac{1}{S}\sum_{j=1}^{S}\sigma_j^2, \tag{A.7}$$

$$\sigma_j^2 = \frac{1}{NM}\sum_{k,l\epsilon\eta} G(k,l)^2 - \mu_j^2, \tag{A.8}$$

$$\mu_j = \frac{1}{NM}\sum_{k,l\epsilon\eta} G(k,l), \tag{A.9}$$

where S represents the image size, and η represents the N-by-M local neighborhood of each pixel in the noisy image G. However, when $\nu^2 \geq \sigma_j^2$ or $\sigma_j^2 = 0$, the Wiener filter returns $F(k,l) = \mu_j$. In this paper, we use the following parameters settings: $N = 3$; $M = 3$.

A.4 Anomalies

After optimizing the datapath of the Wiener filter in fixed-point representation with the proposed video processing validation platform, we obtained results with two different types of anomalies. Some optimized solutions generated by the AWLDT contained negatives pixels while others contained dark pixels, even if the fixed-point solution of the algorithm implementation found by our video processing validation platform meet relatively stringent quality criteria.

In Table A.4, results obtained from the MSE, PSNR and the MSSIM index are reported for nine different images [14]. Each source image contains Gaussian noise with a variance level of 0.01. The noisy images were filtered with the Wiener Filter in floating- and fixed-point resolution. The fixed-point implementation of the Wiener Filter was generated by our video processing validation platform with the MSSIM Index and a performance objective of 0.95. The Wiener Filter optimized fixed-point implementation obtained from the video processing validation platform generates results with visual anomalies. Fig. 3 illustrates an example taken from Table A.4 of a section of the *Lena* filtered image in fixed-point resolution.

Tableau A.1 Comparison of global image metrics for different images containing noise with a variance level of 0.01 and corresponding images processed by the Wiener filter in fixed and floating-point.

Image	**Gaussian Noise**			**Wiener Filter (floating-point)**			**Wiener Filter (fixed-point)**		
	MSE	PSNR	MSSIM	MSE	PSNR	MSSIM	MSE	PSNR	MSSIM
Arctichare	373.29	22.410	0.292278	103.76	27.971	0.732639	141.40	29.626	0.711137
Boat	189.18	25.362	0.644498	81.08	29.041	0.807659	97.51	28.240	0.795193
Cameraman	300.67	23.350	0.541062	112.06	27.636	0.739830	135.10	26.824	0.729231
Cat	207.38	24.963	0.755108	128.70	27.035	0.844317	147.08	26.455	0.832029
Couple	164.78	25.962	0.729978	116.78	27.457	0.767334	128.54	27.041	0.773001
Fruits	273.73	23.758	0.450619	69.81	29.692	0.766572	91.73	28.506	0.745399
Goldhill	145.21	26.511	0.753261	90.32	28.573	0.770426	110.31	27.705	0.754731
Lena	172.32	25.767	0.629845	66.37	29.911	0.811397	87.31	28.720	0.788909
Peppers	166.65	25.913	0.620603	48.42	31.281	0.868091	64.72	30.021	0.849463

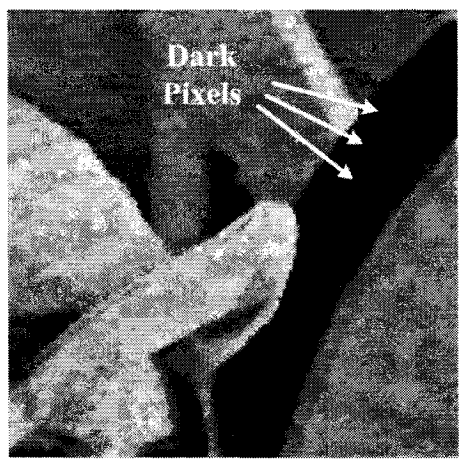


Figure A.3 Section of *Lena* filtered image in fixed-point resolution from Table A.4.

Even if the performance objective is achieved with the Wiener filter fixed-point implementation, the optimal solution generated by the AWLDT contains visual anomalies and that solution should have been rejected. Sections A.4.1 and A.4.2 elaborate on two types of anomalies generated by the video processing validation platform that we wish to avoid.

A.4.1 Negative Pixels

Datapath optimization generates solutions with optimized operands that propagate negative results through the end of the digital filter. Since we use a metric based on an overall score, these few negative pixels have only a small influence over the final score, thus it is possible to obtain a solution satisfying the performance criteria and generating this type of anomaly. Images shown in Fig. A.4 that were obtained from simulations of the Wiener filter processed in fixed-point resolution that reached a performance objective of 0.9832, a relatively high score, exhibit this type of anomaly. The source image contains 85% of JPEG compression noise. The negative pixels are shown as white pixels in the filtered image, see Fig. A.4(b), and are reported as black spots in the SSIM index map; see Fig. A.4(c).

A.4.2 Dark Pixels

We also obtained optimized solutions containing pixels with very low brightness level. Even a small number of such pixels can be detected by a human observer because their brightness differs significantly from the one of their neighbors. Images shown in Fig. A.5 and Fig. A.6 that were obtained from the simulation of an implementation of the Wiener filter processed in fixed-point resolution, and that reaches a performance objective of 0.9832, illustrate this type of anomaly. Each

(a) (b) (c)

Figure A.4 Simulation results with negative pixels. (a) *Lena* filtered image in floating-point, MSSIM = 0.834480; (b) *Lena* filtered image in fixed-point, MSSIM = 0.822677; (c) SSIM map comparing the *Lena* filtered image in floating and fixed-point.

source image contains gaussian noise with a variance level of 0.01. The dark pixels are reported as black spot in the SSIM index map, see Fig. A.5(c) and Fig. A.6(c). The fact that the AWLDT accepts solutions with dark pixels can also be explained by the use of a metric based on an overall score.

A.4.3 Numerical instabilities

These anomalies in the optimal solutions can be explained by the fact that the Wiener filter becomes numerically instable when some operands are optimized too much. In particular, in our implementation of the Wiener filter, one of the operand results consists of computing the operation $G(k,l) - \mu_j$ in A.6. This specific operation may generate negative results that are propagated to three other operands of the filter. These negative results may appear either in floating-point or in fixed-point resolution. However, the result from this operation serves has a numerator inside a division operation. If the value of the denominator is truncated too much in fixed-point resolution, the division opera-tion generates a result much higher

(a) (b) (c)

Figure A.5 Simulation results obtained with dark pixels. (a) *Cameraman* filtered image in floating-point, MSSIM = 0.739830; (b) *Cameraman* filtered image in fixed-point, MSSIM = 0.711923; (c) SSIM map comparing the *Cameraman* filtered image in floating and fixed-point.

than normal. The last operation consists of adding this possibly amplified negative result with the current pixel local mean, which explains the generation of negative pixels. These anomalies in the resulting images motivated us to develop a new OIQM presented in Section A.5 and which is able to automatically detect numerical instabilities inside a video processing algorithm in fixed point resolution.

A.5 Problem Solution

A first solution was used in order to solve the negative pixels anomaly shown in Fig. A.4. Any result produced by the AWLDT that contained negative pixels was automatically rejected. In spite of this new measure, we still encountered results with dark pixels anomalies, see Fig. A.6. In order to remove the dark pixel anomaly, we complemented the SSIM index with an additional test based on a variable called the *Pixel Threshold* (PT). This test sets a threshold for the maximum absolute value of the difference between the brightness of pixels produced by floating- and

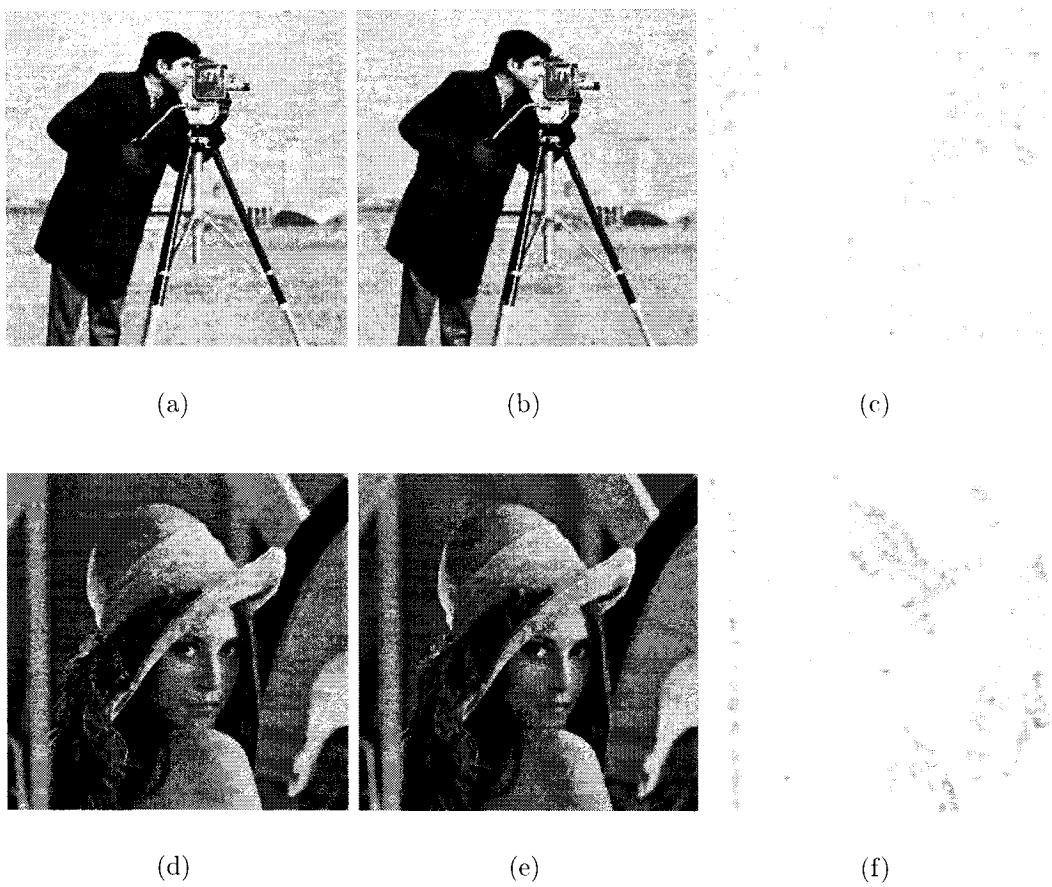
(a) (b) (c)

Figure A.6 Simulation results obtained with dark pixels. (a) *Lena* filtered image in floating-point, MSSIM = 0.811397; (b) *Lena* filtered image in fixed-point, MSSIM = 0.775641; (c) SSIM map comparing the *Lena* filtered image in floating and fixed-point.

a fixed-point processing.

The *Pixel Threshold* is a nonlinear metric providing a local score, which may reject a solution if one pixel of the image does not satisfy the performance criteria. The new OIQM generated from the combination of the global and local metrics removes the numerical instability inside the optimized implementations of video processing algorithm. Fig. A.7 illustrates results obtained from the simulation of the Wiener filter in fixed-point resolution with a performance objective of 0.95 and a maximum pixel threshold of 34. Each source image contains gaussian noise with a variance level of 0.01. It is significant that the images produced by the Wiener filter validated with our new OIQM do not contain anomalies, even if the performance objective is lower than the one used in Fig. A.4 and Fig. A.6.

Fig. A.8 illustrates the same section of "Lena" filtered image in fixed-point resolution as shown in Fig. A.3. This image was obtained with the Wiener filter processed in fixed-point resolution on an implementation derived with the new OIQM combining the global and local metric. These images do not contain dark



(a) (b) (c)

(d) (e) (f)

Figure A.7 Simulation results obtained with the new OIQM. (a) *Cameraman* filtered image in floating-point, MSSIM = 0.739830; (b) *Cameraman* filtered image in fixed-point, MSSIM = 0.729231; (c) SSIM map comparing the *Cameraman* filtered image in floating and fixed-point; (d) *Lena* filtered image in floating-point, MSSIM = 0.811397; (e) *Lena* filtered image in fixed-point, MSSIM = 0.788909; (f) SSIM map comparing the *Lena* filtered image in floating and fixed-point.

pixels, by contrast to images in Fig. A.8. In Table A.5, results obtained with our first solution and our new OIQM are reported for nine different images [14]. In the case of the first solution, the performance objective was 0.9832, whereas with the new OIQM, the performance objective was 0.95 and the maximum pixel threshold objective was 34. Each source image contains gaussian noise with a variance level of 0.01. The results show that according to the global OIQM, the MSSIM in our case, final scores are very similar. However, the observed maximum pixel difference (OMPD) are very different. Results obtained with the new OIQM did not contain visual anomalies (see Fig. A.7) that the first solution still produced (see Fig. A.5 and Fig. A.6).

Figure A.8 Section of *Lena* filtered image in fixed-point resolution without anomalies.

A.6 Conclusion

In this paper, an approach to validate video processing algorithms and their implementations is presented. The aim of this approach is to push the task of video processing algorithm validation at a higher level of abstraction for a hardware designer. In order to achieve this goal, our methodology uses an automatic video

Tableau A.2 Comparison between the first solution and the new OIQM.

Image	First solution		New OIQM	
	MSSIM	OMPD	MSSIM	OMPD
Arctichare	0.702804	48	0.711137	33
Boat	0.776457	51	0.795193	32
Cameraman	0.711923	81	0.729231	31
Cat	0.807311	72	0.832029	33
Couple	0.745483	56	0.773001	27
Fruits	0.731802	69	0.745399	28
Goldhill	0.746624	67	0.754731	28
Lena	0.775641	52	0.788909	26
Peppers	0.835310	43	0.849463	25

processing validation platform that takes advantage of a new objective image quality metric and an AWLDT. The use of objective image quality metrics allows removing human observers from the process of evaluating image quality.

Experimental results demonstrated that a metric combining a global and a local performance objective avoids implementations of video processing algorithms producing large local errors due to numerical instability. The adopted local metric is a pixel to pixel comparison that works well with spatial filter. However, since we want to optimize motion adaptive de-interlacing algorithms, we investigate extensions to the method.

A.7 Acknowledgments

The authors would like to thank Micronet and Gennum Corporation for their financial support, and A. Veenstra and J.-M. Tremblay from Gennum Corporation for their technical guidance.

References

[1] M. Cupak, F. Catthoor, and H.J. De Man. Efficient System-Level Functional Verification Methodology for Multimedia Applications. *IEEE Design and Test of Computers*, vol. 20:56–64, March-April 2003.

[2] I.C. Kraljic, F. S. Verdier, G.M. Quenot, and B. Zavidovique. Investigating Real-Time Validation of Real-Time Image Processing. *IEEE International Workshop on Computer Architecture for Machine Perception*, vol. 20:116–125, October 1997.

[3] M.-A. Cantin, S. Regimbal, S. Catudal, and Y. Savaria. An Unified Environment to Assess Image Quality in Video Processing. *Journal of Circuits, Systems and Computers*, vol. 13, no. 6:1289–1306, December 2004.

[4] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie. An Automatic Word Length Determination Method. *IEEE International Symposium on Circuits and Systems*, vol. 5:53–56, 2001.

[5] M.-A. Cantin, Y. Savaria, and P. Lavoie. A Comparison of Automatic Word Length Optimization Procedures. *IEEE International Symposium on Circuits and Systems*, vol. 2:612–615, 2002.

[6] E.M. Clarcke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, January 2002.

[7] E. Brinksma and A. Mader. Model Checking Embedded System Designs. *Proceeding of the Sixth International Workshop on Discrete Event Systems*, pages 151–158, October 2002.

[8] Z. Wang and A.C. Bovik. A Universal Image Quality Index. *IEEE Signal Processing Letters*, vol. 9, no. 3:81–84, March 2002.

[9] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, vol. 13, no. 4:600–612, April 2004.

[10] B. Furth and O. Marqure. *The Handbook of Video Databases: Design and Applications*. CRC Press, September 2003.

[11] Z. Wang and A.C. Bovik. Why is Image Quality Assessment so Difficult? *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 4:3313–3316, May 2002.

[12] A.M. Eskicioglu and P.S. Fisher. Image Quality Measures and Their Performance. *IEEE Transactions on Communications*, vol. 43, no. 12:2959–2965, December 1995.

[13] J.S. Lim. *Two-Dimensional Signal and Image Processing*. Prentice Hall, 1990.

[14] A.G. Weber. The USC-SIPI Image Database: Version 5. Technical report, Signal and Image Processing, Institute University of Southern California, October 1997.

ANNEXE B

PARAMETERS ESTIMATION APPLIED TO AUTOMATIC VIDEO PROCESSING ALGORITHMS VALIDATION

Serge Catudal, Marc-André Cantin, and Yvon Savaria
Electrical Engineering Department, École Polytechnique de Montréal
C.P. 6079, succursale Centre-ville, Montréal (Québec), Canada, H3C 3A7
Email: {catudal, cantin, savaria}@grm.polymtl.ca

Abstract

An automatic and systematic method to estimate parameters and to validate implementations of video processing algorithms is presented. Correctly adjusting video processing parameters can have a significant impact on the algorithm performance and behavior. Since video processing algorithms parameters are often determined empirically, a new method to automatically estimate parameters is proposed. The method takes advantage of an automatic word length determination tool embedded into a performance driven validation platform that was modified in order to automatically compute optimum parameter values. Experimental results obtained with two spatial filters and an intra-field deinterlacing algorithm are presented to illustrate the validity of our approach.

B.1 Introduction

An abundant number of video processing algorithms are controlled by parameters that must be set prior to the processing. Correctly adjusting these parameters can

have a significant impact on their performance and behavior. Such parameters are often determined empirically by the designer [1][2][3] thus an automatic method to determine and validate parameters that control video processing algorithms is needed.

This paper presents a systematic and automatic parameter determination strategy that has two main objectives. The first objective is to find a set of parameter for an analyzed DSP algorithm, values that maximize the quality of images processed by this algorithm. Finding this parameter set is a complex optimization process. Let us consider an example that helps understand the complexity of that optimization process : the Hybrid filter [2] has 2 parameters to optimize. The first parameter can be set to 60 different values and the second parameter can be set to 2450000 different values. Thus, there are $60 \times 2450000 = 1.47 \times 10^8$ possible sets of parameters to implement the Hybrid filter. Since a software simulation of the Hybrid filter evaluated on four images takes 81.3 seconds to compute on a SPARC SunFire V440 station with 8Go of RAM, it would take 378.97 years to try out all the possible sets in order to determine which combination of parameter values is optimal on the selected small image data base.

The second objective of our systematic optimization and automatic validation process is to find a combination of word lengths that minimizes hardware implementation costs for a given image quality. As for the first objective, it is a necessary step for implementing algorithms in image processing systems. The complexity of this second objective is also a challenging problem. In the Hybrid filter example, there are 23 hardware data paths to optimize, and each of them has 32 different possible configurations (in terms of bits) to implement it. Thus, there are $32^{23} = 4.2 \times 10^{34}$ different ways to implement the considered Hybrid filter. In this case, using the same computer under the same load conditions, 1.1×10^{29} years would be required to exhaustively characterize all possibilities in order to establish which combination

of word lengths is optimal.

Thus, to obtain a solution for these two objectives in a realistic time, a powerful word length and parameter optimization tool is required. A method for automatic sizing of video processing algorithm implementations was developed by the authors [4]. It was then extended to become a performance driven validation methodology applicable to video processing by taking advantage of a new Objective Image Quality Metric (OIQM) [5] and an Automatic Word Length Determination Tool (AWLDT) [6][7].

The purpose of this paper is to propose a new method applicable to video processing that can automatically optimize video processing algorithm parameters while optimizing and validating its hardware implementation. The paper is organized as follows. In Section B.2, a modified Automatic Word Length Determination Tool which is used in our video processing validation platform is presented. Section B.3 briefly reviews the three video processing algorithms used as benchmarks. Section B.4 presents results obtained with our modified video processing validation platform. Finally, Section B.5 formulates conclusions.

B.2 Methodology

The proposed method determines the optimum set of parameter values and combination of word lengths through two optimization phases. During the first phase, the method evaluates the optimal values of the parameters when all operands have floating-point resolution. The second phase searches, through a single process, i) the minimum word length combination that minimizes implementation costs, while meeting image quality targets specified by the user, and ii) the optimum value of control parameters according to the finite precision of the datapath. In

the followings sub-sections, these two phases are presented.

B.2.1 Determining the optimum values of the parameters

The problem of optimizing functions (or DSP algorithms) that contain many parameters according to multiple objectives is well-known in the literature and several methods are proposed to address it [8][9]. From image processing algorithms analyzed in our application, it was observed that the domain of solutions for the parameters of interest does not contain multiple local minima. Thus, a simple search-based approach was selected to find the optimal parameter values. The proposed method would benefit from more complex optimization method such as genetic or simulated annealing techniques if. and only if, multiple local minima exist. Otherwise their use would waste ressources because these techniques are computationally more demanding.

In order to guide the search-based approach toward the optimal solution, two quality indices are computed. The first quality index, Q is computed as follow:

$$Q = \frac{1}{M} \sum_{m=0}^{M-1} \frac{MSSIM\left(I_m^o, I_m^n\right)}{MSSIM\left(I_m^o, I_m^f\right)} \tag{B.1}$$

where M is the number of images processed, I_m^o is the mth original image (I^o), I_m^n is the mth noisy input image (I^n) affected by a given noise level, I_m^f is the mth output image filtered (I^f) by the DSP algorithm, and $MSSIM\left(\cdot\right)$ is an objective image quality metric found in [10].

The second quality index denoted W is computed as follow

$$W = \min_{m=0..M-1} \left\{ \frac{MSSIM\left(I_m^o, I_m^n\right)}{MSSIM\left(I_m^o, I_m^f\right)} \right\} \tag{B.2}$$

where $\min(\cdot)$ is the minimum function.

The goal of the first phase is to find a solution that maximizes the sum of the quality indices with an equal weight on each of them. For each parameter P_l, $l = 0, 1, \ldots, L-1$ where L is the number of parameters to optimize, the user sets the upper $P_l^{\max}$ and the lower $P_l^{\min}$ limit values of each parameter. Furthermore, the user defines the minimum resolution step $S_l^{\min}$ of each parameter.

Our search-based approach proceeds as follows. The optimization process starts with initial parameter values set to $\left(P_l^{\max} - P_l^{\min}\right)/2$. Also, the step size S_l of each operand P_l is set to $\left(P_l^{\max} - P_l^{\min}\right)/K$, where K is the maximum number of steps allowed by the AWLDT. The two quality indices are computed and results are kept for reference. Then, iteratively, each parameter is considered as a candidate to increase or decrease its value by one step S_l when all other parameters remain unchanged. The candidate that offers the best increase of the sum of two indices is retained. When no further improvement is obtained, the step size S_l is reduced by a factor of two and the iterative search continues from the best previously available solution. The search ends when none of the modifications tried increases the sum of the indices, and when each step value S_l is equal to the minimum value specified by the user $S_l^{\min}$.

B.2.2 Determining the optimum set of parameters values and combination of word lengths

Once the optimum set of parameters values is found, the second optimization phase searches a combination of word lengths $\{WL_i\}$ and parameters that minimizes the implementation costs and meets the target image quality specified by the user. This second optimization phase directly exploits the AWLDT presented in [6].

As proposed in [5], the automatic validation of image processing algorithms requires the computation of two quality metrics (i) to obtain good quality images and (ii) to guide the AWLDT toward the optimal solution. The metrics used are the Inverse Ratio (IRATIO) and the Pixel Threshold (PT). The global metric IRATIO, as illustrated in Fig. B.1, computes the ratio of the image quality metric (OIQM as defined in [5]) obtained with finite precision data paths (fixed-point OIQM) over the same metric obtained by the first optimization phase (floating-point OIQM).

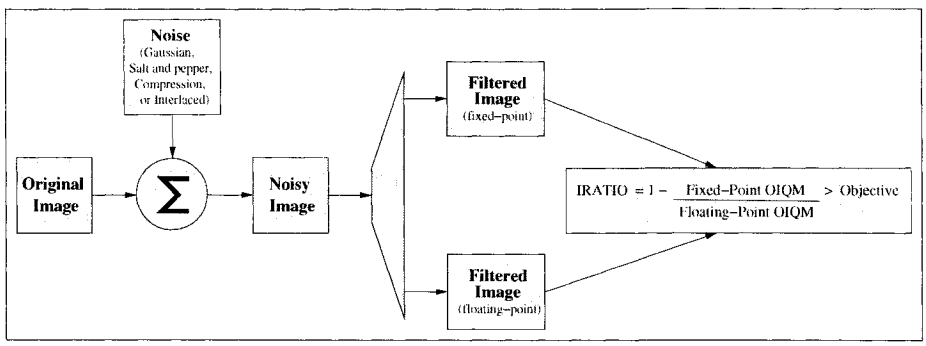


Figure B.1 Flow diagram of the performance assessment method.

The PT metric is computed for each pixel of each image as the maximum absolute difference in intensity between the pixel obtained by the solution of the first optimization phase and the one obtained with finite precision data path. It was shown in [5] that a local metric such as the PT must be computed to avoid visual anomalies. For each metric, the user sets a target quality specification. In [5] the

IRATIO was set to 0.02 and the PT was set to 34.

In the original formulation of the AWLDT in [6], a procedure searches for a combination of minimum word lengths that meets the quality criteria specified by modifying appropriate word lengths. Here, the parameters are treated as operands with the exception that instead of modifying the word length of the operand by adding or removing one bit, the procedure modifies the value of the parameter by increasing and decreasing its value by S_l^{min}. The proposed method optimizes simultaneously (i.e. in a single process) the word lengths of the operands and the values of the parameters defining the algorithm. This allows adjusting precisely the values of these parameters while reducing the error due to the finite precision of the data path. The proposed method was implemented and was successfully applied to three video processing algorithms as presented in the next section.

B.3 Video Processing Algorithms

In order to test our methodology, three different video processing algorithms that are partly defined by parameters were selected as benchmarks. The first and the second algorithm, described in Section B.3.1 and B.3.2 respectively, are spatial filters. The third algorithm, described in Section B.3.3, is a video deinterlacing algorithm.

B.3.1 SUSAN filter

The Smallest Univalue Segment Assimilating Nucleus (SUSAN) spatial filter [1] is described by equation:

$$J(x,y) = \sum_{i \neq 0, j \neq 0} I(x+i, y+j) \cdot S(i,j). \tag{B.3}$$

where

$$S(i,j) = \frac{e^{\frac{i^2+j^2}{2\tau^2} - \frac{(I(x+i,y+j)-I(x,y))^2}{\beta^2}}}{\sum_{i \neq 0, j \neq 0} e^{\frac{i^2+j^2}{2\tau^2} - \frac{(I(x+i,y+j)-I(x,y))^2}{\beta^2}}}, \tag{B.4}$$

and where J is the filtered image and I is the noisy image. The SUSAN noise filtering algorithm is controlled by two parameters. The first parameter, τ in Eq. (B.4), is the spatial control parameter that sets the scale of the spatial smoothing. The second parameter, β in Eq. (B.4), is the brightness control parameter that sets the brightness threshold.

B.3.2 Hybrid filter

The Hybrid filter algorithm [2] comes from the combination of the Minimum Mean Square Error (MMSE) filter [11] and the Sigma filter [12]. The output filtered image g^* of the Hybrid filter is computed as follows:

$$g^*(x,y) = \mu_g + K_g(g(x,y) - \mu_g), \tag{B.5}$$

where K_g is a non-linear gain and μ_g is the local mean around a pixel:

$$K_g = MAX\left(0, \frac{\sigma_g^2 - \sigma_n^2}{\sigma_g^2}\right), \tag{B.6}$$

$$\sigma_g^2 = \frac{1}{MN}\sum_{i,j\epsilon\eta}(W_g(g(x-i,y-j)) - \mu_g)^2, \tag{B.7}$$

$$\mu_g = \frac{1}{MN}\sum_{i,j\epsilon\eta} W_g(g(x-i,y-j)), \tag{B.8}$$

$$W_g = \left\{ \begin{array}{ll} 0 & \text{if } |g(x-i,y-j) - g(x,y)| \geq T_s \\ 1 & \text{if } |g(x-i,y-j) - g(x,y)| < T_s \end{array} \right\}, \tag{B.9}$$

where σ_g^2 represents the local variance around a pixel, W_g represents the filter adaptive segmentation mask, and η represents the N-by-M local neighborhood of each pixel in the noisy image g. In this paper, we use the following parameters settings: $M = N = 7$ fixed to reduce computation time in Section B.4. The noise variance described as σ_g^2 in Eq. (B.6) and the segmentation threshold described as T_s in Eq. (B.9) are two parameters that need to be estimated.

B.3.3 ELA deinterlacing algorithm

The modified Edge-base Line Average (ELA) [3] is a widely used intra-field deinterlacing algorithm. The modified ELA is computed as illustrated in Fig. B.2 and uses two parameters described as T_v and T_d. The ELA algorithm reconstructs line k from lines $k-1$ and $k+1$. It is defined as a function of differences specified by labeled arrows between pixel values in these lines. Parameter T_v is the vertical threshold used in the directional difference labeled c in Fig. B.3. The parameter T_d is the directional difference threshold used for the four directional differences illustrated as a, b, d and e in Fig. B.3.

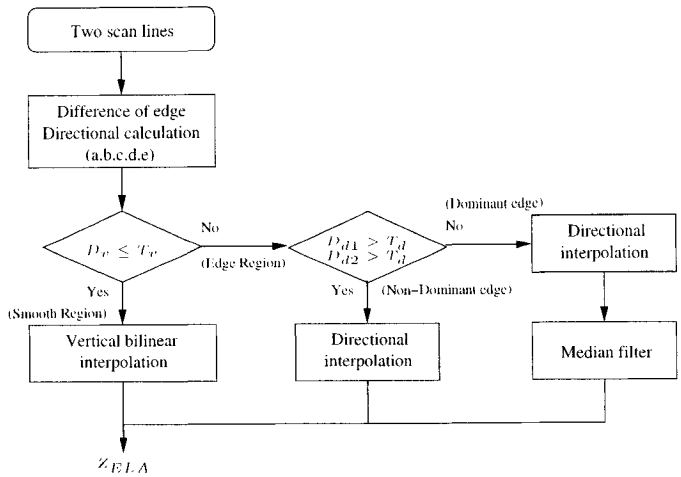


Figure B.2 Flowchart of the modified ELA deinterlacing algorithm.

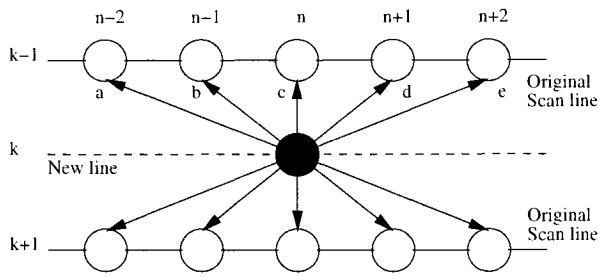


Figure B.3 Pattern of pixels used in the modified ELA algorithm.

The median filter shown in the flowchart of the modified ELA algorithm in Fig. B.2 is computed as follows:

$$Y = Median[X(n, k-1), X(n, k+1), ELA(n, k)] \tag{B.10}$$

B.4 Results

Results obtained with the SUSAN filter, the Hybrid filter and the ELA deinterlacing algorithm are reported in Table B.1. The results from the SUSAN filter were obtained for nine different images from the image database of [13] and the algorithm

has two parameters to estimate and seven operands to optimize. The results from the Hybrid filter were obtained for four different images from the image database of [13] and the algorithm has two parameters to estimate and twenty-three operands to optimize. Finally, the results from the ELA deinterlacing algorithm were obtained with the eighth frame of the *Football* video sequence. The ELA deinterlacing algorithm has two parameters to estimate and four operands to optimize.

Tableau B.1 Results obtained for three different video processing algorithms.

Algorithms	Parameters to estimate	Operands to optimize	HW Cost units	Performance Objective		Performance Obtained		Computational time (s)
				IRATIO	PT	IRATIO	PT	
Hybrid filter	2	23	140	0.03000	34	0.025248	28	505605
SUSAN filter	2	7	65	0.05000	34	0.003128	31	17040
ELA deinterlacer	2	4	23	0.05000	34	0.035429	29	576

The parameter estimations obtained with our video processing validation platform for the Hybrid filter algorithm were $T_s = 46$ and $\sigma_n^2 = 186.845776$. Fig. B.4 shows the contour curves of the quality index Q (see Eq. B.1) for the Hybrid filter parameters. These curves were obtained manually from C/C++ simulations of the Hybrid filter in floating-point representation by setting the parameter T_s in the range $[15, 75]$, with a step of 2, and by setting the parameter σ_n^2 in the range $[10, 255]$, with a step of 5. A total of 1550 simulations of the algorithm were obtained for each image. Fig. B.4 was computed in MATLAB by interpolating the feasible solutions to produce contour plot that are easier to visualize. To conclude this section, we can observe in Table B.1 that the computational time for validating the video processing algorithm implementation is dependent on the number of operands that need to be optimized and that the performance are met with a comfortable margin.

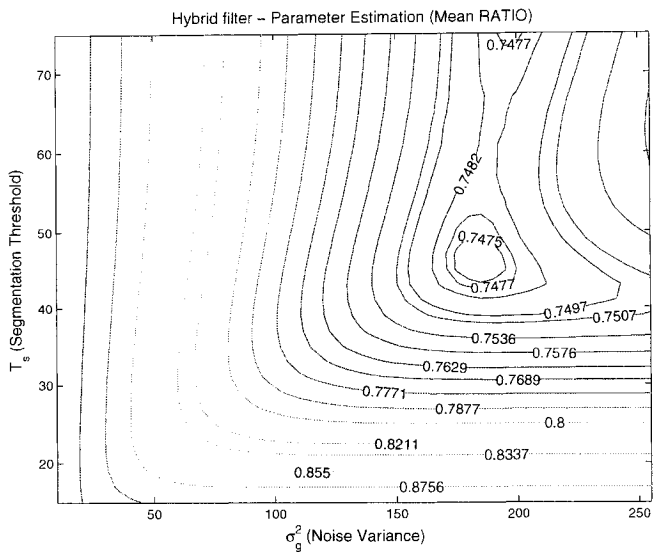


Figure B.4 Quality index Q of the Hybrid filter parameter estimation.

B.5 Conclusion

In this paper, an approach to estimate video processing algorithm parameters and to validate their implementations is presented. The aim of this approach is to push the task of video processing algorithm parameter estimation and implementation validation to a higher level of abstraction for a hardware designer. In order to achieve this goal, a methodology, which takes advantage of an Automatic Word Length Determination Tool (AWLDT) and an Objective Image Quality Metric (OIQM), was proposed.

The use of the OIQM allows the removal of a human observer from the process of evaluating image quality, and guiding the AWLDT toward the optimal solution. The AWLDT tool was modified in order to automatically determine, in a single optimization process, the minimum word length combination that minimizes implementation costs, while meeting image quality targets specified by the user, and to determine the optimum value of control parameters according to the finite

precision of the data path.

The systematic optimization and automatic validation methodology was applied on three image processing algorithms. Experimental results obtained with these algorithms show that the set of parameters obtained from the optimization process concurs with the set of optimal values. They also demonstrate that the proposed methodology can automatically determine and validate parameters that regulate video processing algorithms.

References

[1] S.M. Smith and J.M. Brady. SUSAN - A New Approach to Low Level Image Processing. *International Journal of Computer Vision*, vol. 23:45–78, 1997.

[2] L. Loiseau. Méthode de Conception pour la Réutilisation et Optimisation Architecturale Appliqueées à un Réducteur de Bruit Vidéo. Master's thesis, École Polytechnique de Montréal, 2001.

[3] H.Y. Lee, J.W. Park, T.M. Bae, S.U. Choi, and Y.H. Ha. Adaptive Scan Rate Up-Conversion System Based on Human Visual Characteristics. *IEEE Transactions on Consumer Electronics*, vol. 46, no. 4:999–1006, 2000.

[4] M.-A. Cantin, S. Regimbal, S. Catudal, and Y. Savaria. An Unified Environment to Assess Image Quality in Video Processing. *Journal of Circuits, Systems and Computers*, vol. 13, no. 6:1289–1306, December 2004.

[5] S. Catudal, M.-A. Cantin, and Y. Savaria. Performance Driven Validation Applied to Video Processing. *WSEAS Transaction on Electronics*, vol. 1, no. 3:568–575, July 2004.

[6] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie. An Automatic Word Length Determination Method. *IEEE International Symposium on Circuits and Systems*, vol. 5:53–56, 2001.

[7] M.-A. Cantin, Y. Savaria, and P. Lavoie. A Comparison of Automatic Word Length Optimization Procedures. *IEEE International Symposium on Circuits and Systems*, vol. 2:612–615, 2002.

[8] C. Onwubiki. *Introduction to Engineering Design Optimization.* Prentice Hall, 2000.

[9] A.D. Belegundu. *Optimization Concepts and Applications in Engineering.* Prentice Hall, 1999.

[10] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, vol. 13, no. 4:600–612, April 2004.

[11] J.S. Lim. *Two-Dimensional Signal and Image Processing.* Prentice Hall, 1990.

[12] J.S. Lee. *Digital Image Smoothing and the Sigma Filter*, volume vol. 24. Computer Vision, Graphics and Image Processing, 1983.

[13] A.G. Weber. The USC-SIPI Image Database: Version 5. Technical report, Signal and Image Processing, Institute University of Southern California, October 1997.

ANNEXE C

CODE SOURCE DE LA MÉTRIQUE SSIM INDEX

C.1 Fichier ssim_index.h

```
//-----------------------------------------------------------------------------
// Title          : SSIM Index
// Project        : Video Processing Module
//-----------------------------------------------------------------------------
// File           : ssim_index.h
// Author         : Serge Catudal
// Created        : 17/11/2003
// Last modified  : 28/01/2004
//-----------------------------------------------------------------------------
// Description    : Structural SIMilarity Index
//-----------------------------------------------------------------------------
// Copyright (c) 2003 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 17/11/2003 : created
//-----------------------------------------------------------------------------

/** @defgroup oiqm_pack OIQM : Objective Image Quality Metric
 *  This library is the OIQM library utility.
 *  @ingroup vpve_core
 *  @author Serge Catudal
 *  @version 1.0
 *  @date 28/01/2004
 */

#ifndef _SSIM_INDEX_
#define _SSIM_INDEX_

// STL inclusions
#include <string>

// C/C++ inclusions
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// POSIX.4 inclusions
#include <sys/types.h>
#include <sys/wait.h>

using namespace std;

// Size of the circular symmetric gaussian weighting function
#define size_window 11

// Constants in the SSIM index formula
#define K1 0.01
#define K2 0.03
#define L   255

// Circular symmetric gaussian weighting function
const double window[size_window][size_window] = {
   {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000},
   {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
   {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
   {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
   {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
   {0.0003,0.0020,0.0096,0.0291,0.0567,0.0708,0.0567,0.0291,0.0096,0.0020,0.0003},
   {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
   {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
   {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
   {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
```

```
  {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000}
};

/**
 *  @ingroup oiqm_pack
 *  @brief This class is the Structural SIMilarity Index
 *  @author Serge Catudal
 *  @version 1.0
 *  @date 04/12/2003
 */
class ssim_index{
 public:
  /// Constructor
  ssim_index();
  /// Copy constructor
  ssim_index(const ssim_index&);
  /// Overload the equal (=) operator
  ssim_index& operator=(const ssim_index&);
  /// Destructor
  ~ssim_index();

  // Access functions
  double** get_ssim_map(void)       {return ssim_map;}
  void     set_image_height(int h) {height = h;}
  void     set_image_width(int w)  {width  = w;}
  void     set_ssim_map(void);

  // Utility functions
  double   compute_ssim_index(int**,int**);
  void     write_ssim_map(string);

 private:
  /// First time in creating the SSIM Map
  bool    first_time;
  /// Image height
  int     height;
  /// SSIM map
  double **ssim_map;
  /// SSIM map
  int     **ssim_map_img;
  /// Image width
  int     width;
};

#endif
```

C.2 Fichier ssim_index.cc

```
//-----------------------------------------------------------------------------
// Title          : SSIM Index
// Project        : Video Processing Module
//-----------------------------------------------------------------------------
// File           : ssim_index.cc
// Author         : Serge Catudal
// Created        : 17/11/2003
// Last modified : 28/01/2004
//-----------------------------------------------------------------------------
// Description    : Structural SIMilarity Index
//-----------------------------------------------------------------------------
// Copyright (c) 2003 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 17/11/2003 : created
//-----------------------------------------------------------------------------

#include <ssim_index.h>

//-----------------------------------------------------------------------------
// Default Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
ssim_index::ssim_index(){
  first_time = true;
}

//-----------------------------------------------------------------------------
// Copy Constructor
/**
```

```
 * @brief
 */
//-------------------------------------------------------------------------
ssim_index::ssim_index(const ssim_index& I){
  first_time    = I.first_time;
  height        = I.height;
  width         = I.width;
  ssim_map      = I.ssim_map;
  ssim_map_img = I.ssim_map_img;
}

//-------------------------------------------------------------------------
// = Operator Overloading
/**
 * @brief
 */
//-------------------------------------------------------------------------
ssim_index& ssim_index::operator=(const ssim_index& I){
  if(this != &I){
    first_time    = I.first_time;
    height        = I.height;
    width         = I.width;
    ssim_map      = I.ssim_map;
    ssim_map_img = I.ssim_map_img;
  }
  return *this;
}

//-------------------------------------------------------------------------
// Destructor
/**
 * @brief
 */
//-------------------------------------------------------------------------
ssim_index::~ssim_index(){
  if(!first_time){
    for(int j=0; j<((height - size_window) + 1); j++){
      delete[] ssim_map[j];
      delete[] ssim_map_img[j];
    }
    delete[] ssim_map;
    delete[] ssim_map_img;
    ssim_map      = NULL;
    ssim_map_img = NULL;
  }
}

//-------------------------------------------------------------------------
// Access functions
//-------------------------------------------------------------------------
/**
 * @brief Set the Structural SIMilarity Index Map
 */
void ssim_index::set_ssim_map(void){
  int j;

  // Free the memory
  if(!first_time){
    for(int j=0; j<((height - size_window) + 1); j++){
      delete[] ssim_map[j];
      delete[] ssim_map_img[j];
    }
    delete[] ssim_map;
    delete[] ssim_map_img;
    ssim_map      = NULL;
    ssim_map_img = NULL;
  } else {
    first_time = false;
  }

  ssim_map      = new double*[(height - size_window) + 1];
  ssim_map_img = new int*[(height - size_window) + 1];
  for(j = 0; j < (height - size_window) + 1; j++){
    ssim_map[j]      = new double[(width - size_window) + 1];
    ssim_map_img[j] = new int[(width - size_window) + 1];
  }
}

//-------------------------------------------------------------------------
// Utility functions
//-------------------------------------------------------------------------
/**
 * @brief Compute the Structural SIMilarity Index
 */
double ssim_index::compute_ssim_index(int** original, int** distorted){
  int    border;
```

```
double C1,C2;
double denominator1,denominator2;
double numerator1,numerator2;
double numerator3,denominator3;
int    i,j;
int    max_ssim_map = 0;
int    m,n;
int    **img1_img2,**img1_sq,**img2_sq;
double **mu1_mu2,**mu1,**mu2;
double **mu1_sq,**mu2_sq;
double **sigma12,**sigma1_sq,**sigma2_sq;
double result = 0.0;
double contrast;
double mean_contrast = 0.0;
double luminance;
double mean_luminance = 0.0;
double structure;
double mean_structure = 0.0;

// Variable initialization
border = ((size_window - 1) / 2);

// Constants initialiazation
C1 = (K1*L)*(K1*L);
C2 = (K2*L)*(K2*L);

// Calculate the square for the original and distorted image
img1_img2 = new int*[height];
img1_sq   = new int*[height];
img2_sq   = new int*[height];
for(j = 0; j < height; j++){
  img1_img2[j] = new int[width];
  img1_sq[j]   = new int[width];
  img2_sq[j]   = new int[width];
  for(i = 0; i < width; i++){
    img1_img2[j][i] = original[j][i] * distorted[j][i];
    img1_sq[j][i]   = original[j][i] * original[j][i];
    img2_sq[j][i]   = distorted[j][i] * distorted[j][i];
  }
}

// Apply the 2D Filter on the mean intensity and the variance
mu1       = new double*[(height - size_window) + 1];
mu2       = new double*[(height - size_window) + 1];
sigma12   = new double*[(height - size_window) + 1];
sigma1_sq = new double*[(height - size_window) + 1];
sigma2_sq = new double*[(height - size_window) + 1];
mu1_mu2   = new double*[(height - size_window) + 1];
mu1_sq    = new double*[(height - size_window) + 1];
mu2_sq    = new double*[(height - size_window) + 1];
for(j = (0 + border); j < (height - border); j++){
  mu1[j-border]       = new double[(width - size_window) + 1];
  mu2[j-border]       = new double[(width - size_window) + 1];
  sigma12[j-border]   = new double[(width - size_window) + 1];
  sigma1_sq[j-border] = new double[(width - size_window) + 1];
  sigma2_sq[j-border] = new double[(width - size_window) + 1];
  mu1_mu2[j-border]   = new double[(width - size_window) + 1];
  mu1_sq[j-border]    = new double[(width - size_window) + 1];
  mu2_sq[j-border]    = new double[(width - size_window) + 1];
  for(i = (0 + border); i < (width - border); i++){
    mu1[j-border][i-border]       = 0;
    mu2[j-border][i-border]       = 0;
    sigma12[j-border][i-border]   = 0;
    sigma1_sq[j-border][i-border] = 0;
    sigma2_sq[j-border][i-border] = 0;
    // Apply the circular symmetric gaussian weighting function on each pixel
    for(n = (0 - border); n < (size_window - border); n++){
      for(m = (0 - border); m < (size_window - border); m++){
        mu1[j-border][i-border] =
          mu1[j-border][i-border] +
          ((double)(original[j+n][i+m]) * window[n+border][m+border]);
        mu2[j-border][i-border] =
          mu2[j-border][i-border] +
          ((double)(distorted[j+n][i+m]) * window[n+border][m+border]);
        sigma12[j-border][i-border] =
          sigma12[j-border][i-border] +
          ((double)(img1_img2[j+n][i+m]) * window[n+border][m+border]);
        sigma1_sq[j-border][i-border] =
          sigma1_sq[j-border][i-border] +
          ((double)(img1_sq[j+n][i+m]) * window[n+border][m+border]);
        sigma2_sq[j-border][i-border] =
          sigma2_sq[j-border][i-border] +
          ((double)(img2_sq[j+n][i+m]) * window[n+border][m+border]);
      }
    }
```

```
      // SSIM(x,y) = [l(x,y)] * [c(x,y)] * [s(x,y)]
      //
      //     where
      //
      //           l(x,y) is the luminance comparison
      //           l(x,y) = (2*mu_x*mu_y + C1) / (mu_x^2 + mu_y^2 + C1)
      //
      //           c(x,y) is the contrast comparison
      //           c(x,y) = (2*sigma_x*sigma_y + C2) / (sigma_x^2 + sigma_y^2 + C2)
      //
      //           s(x,y) is the structure comparison
      //           s(x,y) = (sigma_xy + C3) / (sigma_x*sigma_y + C3)
      //
      // SSIM(x,y) = [(2*mu_x*mu_y + C1) * (2*sigma_x*sigma_y + C2)] /
      //             [(mu_x^2 + mu_y^2 + C1) * (sigma_x^2 + sigma_y^2 + C2)]
      //
      mu1_mu2[j-border][i-border] = mu1[j-border][i-border] * mu2[j-border][i-border];
      mu1_sq[j-border][i-border] = mu1[j-border][i-border] * mu1[j-border][i-border];
      mu2_sq[j-border][i-border] = mu2[j-border][i-border] * mu2[j-border][i-border];
      sigma12[j-border][i-border] = sigma12[j-border][i-border] - mu1_mu2[j-border][i-border];
      sigma1_sq[j-border][i-border] = sigma1_sq[j-border][i-border] -
                                      mu1_sq[j-border][i-border];
      sigma2_sq[j-border][i-border] = sigma2_sq[j-border][i-border] -
                                      mu2_sq[j-border][i-border];
      numerator1 = ((2 * mu1_mu2[j-border][i-border]) + C1);
      numerator2 = ((2 * sigma12[j-border][i-border]) + C2);
      denominator1 = (mu1_sq[j-border][i-border] + mu2_sq[j-border][i-border] + C1);
      denominator2 = (sigma1_sq[j-border][i-border] + sigma2_sq[j-border][i-border] + C2);
      ssim_map[j-border][i-border] = (numerator1 * numerator2) / (denominator1 * denominator2);
      ssim_map_img[j-border][i-border] = (int)(255*ssim_map[j-border][i-border]);
      if(ssim_map_img[j-border][i-border] < 0)
        ssim_map_img[j-border][i-border] = 0;
      else if (ssim_map_img[j-border][i-border] > 255)
        ssim_map_img[j-border][i-border] = 255;

      if(255-ssim_map_img[j-border][i-border] > max_ssim_map)
        max_ssim_map = 255-ssim_map_img[j-border][i-border];

      result = result + ssim_map[j-border][i-border];
    }
  }

  // A mean SSIM index (MSSIM) is used to evaluate the overall image quality
  //
  // MSSIM(X,Y) = (1/M)SUM(SSIM(x,y))
  //
  result = result / (((height - size_window) + 1) * ((width - size_window) + 1));

  // Free the memory
  for(j = 0; j < height; j++){
    delete[] img1_img2[j];
    delete[] img1_sq[j];
    delete[] img2_sq[j];
    if((j > (0 + border - 1)) && (j < (height - border))){
      delete[] mu1_mu2[j-border];
      delete[] mu1[j-border];
      delete[] mu2[j-border];
      delete[] mu1_sq[j-border];
      delete[] mu2_sq[j-border];
      delete[] sigma12[j-border];
      delete[] sigma1_sq[j-border];
      delete[] sigma2_sq[j-border];
    }
  }
  delete []img1_img2;
  delete []img1_sq;
  delete []img2_sq;
  delete []mu1_mu2;
  delete []mu1;
  delete []mu2;
  delete []mu1_sq;
  delete []mu2_sq;
  delete []sigma12;
  delete []sigma1_sq;
  delete []sigma2_sq;
  img1_img2 = NULL;
  img1_sq   = NULL;
  img2_sq   = NULL;
  mu1_mu2   = NULL;
  mu1       = NULL;
  mu2       = NULL;
  mu1_sq    = NULL;
  mu2_sq    = NULL;
  sigma12   = NULL;
  sigma1_sq = NULL;
  sigma2_sq = NULL;
```

```
  return result;
}

/**
 * @brief Write the SSIM map has a Raw PGM image file
 */
void ssim_index::write_ssim_map(string filename){
  ofstream EcrireBin;

  char          *ctmp = new char[10];
  int           i,j;
  unsigned int  itr;

  // Create raw PGM image file header
  char* header_tmp = new char[20];
  strcat(header_tmp,"P5\n");
  sprintf(ctmp,"%d",((width - size_window) + 1));
  strcat(header_tmp,ctmp);
  strcat(header_tmp," ");
  sprintf(ctmp,"%d",((height - size_window) + 1));
  strcat(header_tmp,ctmp);
  strcat(header_tmp,"\n255\n");

  // Write raw PGM image file pixel
  EcrireBin.open(filename.c_str(), ios::binary);
  EcrireBin.seekp(0,ios::beg);
  for(itr=0; itr<strlen(header_tmp); itr++)
    EcrireBin.write((char *) &header_tmp[itr],sizeof(char));
  for(j = 0; j < ((height - size_window) + 1); j++){
    for(i = 0; i < ((width - size_window) + 1); i++){
      EcrireBin.write((char *) &(unsigned char)ssim_map_img[j][i],sizeof(unsigned char));
    }
  }

  EcrireBin.close();

  delete ctmp;
}
```

ANNEXE D

RÉDUCTEUR DE BRUIT VIDÉO

D.1 Code source du filtre Hybride

D.1.1 Fichier hybrid_filter.h

```
//-----------------------------------------------------------------------------
// Title          : Hybrid image filter
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : hybrid_filter.h
// Author         : Serge CATUDAL
// Created        : 26/04/2004
// Last modified  : 05/05/2004
//-----------------------------------------------------------------------------
// Description    : Combination of the Sigma filter with the MMSE filter
//-----------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 26/04/2004 : created
//-----------------------------------------------------------------------------

#ifndef _HYBRID_FILTER_HEADER_
#define _HYBRID_FILTER_HEADER_

// VPVE inclusions
#include <image_filter.h>

/**
 *  @ingroup filter_pack
 *  @brief This class is the Hybrid Filter
 *  @author Serge Catudal
 *  @version 1.0
 *  @date 26/04/2004
 */

class hybrid_filter : public image_filter {
 public:
  /// Constructor
  hybrid_filter();
  /// Copy Constructor
  hybrid_filter(const hybrid_filter&);
  /// Overload the equal (=) operator
  hybrid_filter& operator=(const hybrid_filter&);
  /// Destructor
  ~hybrid_filter();

  // Access functions
  void set_mask_size_x(int x) {mask_size_x = x;}
  void set_mask_size_y(int y) {mask_size_y = y;}
  void set_nu(float ne)       {nu          = ne;}
  void set_threshold(int t)   {threshold   = t;}

  // Hybrid filter functions
  int** apply_filter(int **img_in);
  void  generate_matlab_file(string parameter_filename,string matlab_filename,
                             int** img_src, int** img_noisy, float noisy_quality);

  // Other functions
  float compute_mean(int**);
  float compute_variance(int**,float);

 private:
```

```
  /// Mask size width
  int          mask_size_x;
  /// Mask size height
  int          mask_size_y;
  /// MSSIM index
  ssim_index *my_ssim_index;
  /// Noise Estimation
  float        nu;
  /// Brightness threshold
  int          threshold;
};

#endif
```

D.1.2 Fichier hybrid_filter.cc

```
//-----------------------------------------------------------------------------
// Title          : Hybrid image filter
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : hybrid_filter.cc
// Author         : Serge CATUDAL
// Created        : 26/04/2004
// Last modified  : 05/05/2004
//-----------------------------------------------------------------------------
// Description    : Combination of the Sigma filter with the MMSE filter
//-----------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 26/04/2004 : created
//-----------------------------------------------------------------------------

#include <hybrid_filter.h>

//-----------------------------------------------------------------------------
// Default Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
hybrid_filter::hybrid_filter() : image_filter(VPVE_HYBRID_FILTER){
  mask_size_x    = 0;
  mask_size_y    = 0;
  nu             = 0;
  threshold      = 0;
  my_ssim_index = new ssim_index();
}

//-----------------------------------------------------------------------------
// Copy Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
hybrid_filter::hybrid_filter(const hybrid_filter& hf) : image_filter(hf){
  mask_size_x = hf.mask_size_x;
  mask_size_y = hf.mask_size_y;
  nu          = hf.nu;
  threshold   = hf.threshold;
}

//-----------------------------------------------------------------------------
// = Operator Overloading
/**
 * @brief
 */
//-----------------------------------------------------------------------------
hybrid_filter& hybrid_filter::operator=(const hybrid_filter& hf){
  if(this != &hf){
    mask_size_x = hf.mask_size_x;
    mask_size_y = hf.mask_size_y;
    nu          = hf.nu;
    threshold   = hf.threshold;
  }
  return *this;
}

//-----------------------------------------------------------------------------
// Destructor
```

```
/**
 * @brief
 */
//------------------------------------------------------------------------------
hybrid_filter::~hybrid_filter(){
  delete my_ssim_index;
}

//------------------------------------------------------------------------------
// Hybrid Filter Functions
//------------------------------------------------------------------------------
/**
 * @brief Apply the Hybrid Filter
 */
int** hybrid_filter::apply_filter(int** img_in){
  double **mu;
  double **sigma;
  double K;
  int    **enlarge_img;
  int    **img_out;
  int    **seg;
  int    image_size;
  int    i,j;
  int    x,y;
  int    nb_active_pix;

  // Filtered image initialization
  img_out = new int*[img_height];
  mu      = new double*[img_height];
  sigma   = new double*[img_height];
  for(j=0; j<img_height; j++){
    img_out[j] = new int[img_width];
    mu[j]      = new double[img_width];
    sigma[j]   = new double[img_width];
    for(i=0; i<img_width; i++){
      img_out[j][i] = 0;
      mu[j][i]      = 0;
      sigma[j][i]   = 0;
    }
  }
  seg = new int*[(2*mask_size_y)+1];
  for(j=0; j<((2*mask_size_y)+1); j++){
    seg[j] = new int[(2*mask_size_x)+1];
    for(i=0; i<((2*mask_size_x)+1); i++)
      seg[j][i] = 0;
  }

  // Variables initialization
  image_size = img_height * img_width;

  // Enlarge img_in
  enlarge_img = new int*[img_height + (2*mask_size_y)];
  for(j=0; j<img_height + (2*mask_size_y); j++)
    enlarge_img[j] = new int[img_width + (2*mask_size_y)];

  for(j=mask_size_y; j<(img_height + mask_size_y); j++)  // Copy img_in
    for(i=mask_size_x; i<(img_width + mask_size_x); i++)
      enlarge_img[j][i] = img_in[j - mask_size_y][i - mask_size_x];
  for(j=0; j<mask_size_y; j++)  // Copy top and bottom rows
    for(i=mask_size_x; i<(img_width + mask_size_x); i++){
      enlarge_img[j][i] = enlarge_img[((2*mask_size_y) + 1)-j][i];
      enlarge_img[j+img_height+mask_size_y][i] =
        enlarge_img[(img_height + mask_size_y - 2)-j][i];
    }
  for(i=0; i<mask_size_x; i++) // Copy left and right columns
    for(j=0; j<(img_height + (2*mask_size_y)); j++){
      enlarge_img[j][i] = enlarge_img[j][((2*mask_size_x) + 1)-i];
      enlarge_img[j][i+img_width+mask_size_x] =
        enlarge_img[j][(img_width + mask_size_x - 2)-i];
    }

  // Compute the Segmentation, Local Mean, Local Variance
  // and the Noise Estimation
  for(j=mask_size_y; j<(img_height + mask_size_y); j++){
    for(i=mask_size_x; i<(img_width + mask_size_x); i++){

      // Segmentation
      nb_active_pix = 0;
      for(y = -mask_size_y; y <= mask_size_y; y++)
        for(x = -mask_size_x; x <= mask_size_x; x++){
          if(abs(enlarge_img[j][i]-enlarge_img[j+y][i+x]) >= threshold){
            seg[y + mask_size_y][x + mask_size_x] = 0;
          } else {
            seg[y + mask_size_y][x + mask_size_x] = 1;
            nb_active_pix++;
          }
```

```
        }

      // Local Mean
      for(y = -mask_size_y; y <= mask_size_y; y++)
        for(x = -mask_size_x; x <= mask_size_x; x++)
          mu[j - mask_size_y][i - mask_size_x] +=
            (seg[y + mask_size_y][x + mask_size_x] *
             enlarge_img[j + y][i + x]);
      mu[j - mask_size_y][i - mask_size_x] =
        mu[j - mask_size_y][i - mask_size_x] / nb_active_pix;

      // Local Variance
      for(y = -mask_size_y; y <= mask_size_y; y++)
        for(x = -mask_size_x; x <= mask_size_x; x++)
          sigma[j - mask_size_y][i - mask_size_x] +=
            (seg[y + mask_size_y][x + mask_size_x] *
             enlarge_img[j + y][i + x] * enlarge_img[j + y][i + x]);
      sigma[j - mask_size_y][i - mask_size_x] =
        ((sigma[j - mask_size_y][i - mask_size_x] / nb_active_pix) -
         (mu[j - mask_size_y][i - mask_size_x] *
          mu[j - mask_size_y][i - mask_size_x]));
    }
  }

  // Final computation
  for(j=0; j<img_height; j++){
    for(i=0; i<img_width; i++){
      // K(x,y) = MAX(0, (sigma(x,y) - nu(x,y))) / sigma(x,y)
      K = sigma[j][i] - nu;
      if(K < 0)
        K = 0;
      else
        K = K / sigma[j][i];

      // g*(x,y) = mu(x,y) + K(x,y)*(g(x,y) - mu(x,y))
      img_out[j][i] = (int)(mu[j][i] + (K * (img_in[j][i] - mu[j][i])));
    }
  }

  // Free the memory
  for(j=0; j<img_height + (2*mask_size_y); j++)
    delete[] enlarge_img[j];
  for(j=0; j<((2*mask_size_y)+1); j++)
    delete[] seg[j];
  for(j=0; j<img_height; j++){
    delete[] mu[j];
    delete[] sigma[j];
  }
  delete[] enlarge_img;
  delete[] seg;
  delete[] mu;
  delete[] sigma;
  enlarge_img = NULL;
  seg         = NULL;
  mu          = NULL;
  sigma       = NULL;

  return img_out;
}

/**
 * @brief Generate the Hybrid filter matlab files
 */
void hybrid_filter::generate_matlab_file(string parameter_filename,
                                         string matlab_filename,
                                         int** img_src,
                                         int** img_noisy,
                                         float noisy_quality){
  char        *filename;
  float       d_t, dt_start, dt_end;
  float       img_quality = 0;
  int         b_t, bt_start, bt_end;
  int         j;
  int         **img_filtered;
  string      tmp_filename;

  ifstream ReadTXT;
  ofstream WriteMatlab_results;

  // Parameter file content:
  //
  // Image Filename
  // Brightness Threshold start
  // Brightness Threshold end
  // Noise estimation start
  // Noise estimation end
```

```
//
// Example of the content of the parameter file:
//
// lena_256.pgm                                          <- Image filename
// 15                                                    <- Threshold start
// 75                                                    <- Threshold end
// 10                                                    <- NU start
// 160                                                   <- NU end

cout << "Reading Parameters:" << endl;

// Open parameter file
ReadTXT.open(parameter_filename.c_str());

// Save matlab results file
filename = new char[40];
strcat(filename,matlab_filename.c_str());
strcat(filename,"_results.m");
WriteMatlab_results.open(filename);
delete[] filename;
filename = NULL;

if(!ReadTXT.fail() && !WriteMatlab_results.fail()){
  // Load the source image
  ReadTXT >> tmp_filename;

  // Load Segmentation threshold parameters
  ReadTXT >> bt_start >> bt_end;
  cout << "\tSegmentation threshold parameters:" << endl;
  cout << "\t\tStart: " << bt_start << endl;
  cout << "\t\tEnd  : " << bt_end << endl;

  // Load NU parameters
  ReadTXT >> dt_start >> dt_end;
  cout << "\tNoise estimation parameters:" << endl;
  cout << "\t\tStart: " << dt_start << endl;
  cout << "\t\tEnd  : " << dt_end << endl;
  cout << "-------------------------------------------------------------------\n";

  // Initialize the ssim_index object
  my_ssim_index->set_image_height(img_height);
  my_ssim_index->set_image_width(img_width);
  my_ssim_index->set_ssim_map();

  // Initialize the filter
  set_mask_size_x(3);
  set_mask_size_y(3);

  // Filter the image with different Hybrid filter parameters
  WriteMatlab_results << "hybrid_filter=[";
  for(b_t = bt_start; b_t <= bt_end; b_t = b_t + 2){
    for(d_t = dt_start; d_t <= dt_end; d_t = d_t + 5){
      set_threshold(b_t);
      set_nu(d_t);
      img_filtered = apply_filter(img_noisy);
      img_quality  = (float)(my_ssim_index->compute_ssim_index(img_src,
                                                               img_filtered));
      if(d_t != dt_end)
        WriteMatlab_results << noisy_quality/img_quality << ",";
      else
        WriteMatlab_results << noisy_quality/img_quality;

      // Free the memory
      for(j=0; j<img_height; j++)
        delete[] img_filtered[j];
      delete[] img_filtered;
      img_filtered = NULL;
    }
    if(b_t != bt_end)
      WriteMatlab_results << ".\n";
  }
  WriteMatlab_results << "];";
}

ReadTXT.close();
WriteMatlab_results.close();
}

//-------------------------------------------------------------------------
// Other functions
//-------------------------------------------------------------------------
/**
 * @brief Compute the Mean of the image
 */
float hybrid_filter::compute_mean(int **img_src){
  double result_mean = 0;
```

```
  int j,i;

  for(j=0; j<img_height; j++)
    for(i=0; i<img_width; i++)
      result_mean += img_src[j][i];
  result_mean = result_mean / (img_height * img_width);

  return (float)result_mean;
}

/**
 * @brief Compute the Variance of the image
 */
float hybrid_filter::compute_variance(int **img_src, float mean){
  double result_variance = 0;
  int j,i;

  for(j=0; j<img_height; j++)
    for(i=0; i<img_width; i++)
      result_variance += (img_src[j][i] * img_src[j][i]);
  result_variance = (result_variance - (mean * mean)) / (img_height * img_width);

  return (float)result_variance;
}
```

D.2 Code source du filtre SUSAN

D.2.1 Fichier susan_filter.h

```
//-----------------------------------------------------------------------------
// Title          : SUSAN image filter
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : susan_filter.h
// Author         : Serge CATUDAL
// Created        : 05/03/2004
// Last modified : 29/04/2004
//-----------------------------------------------------------------------------
// Description    : Smallest Univalue Segment Assimilating Nucleus (SUSAN)
//-----------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 05/03/2004 : created
//-----------------------------------------------------------------------------

#ifndef _SUSAN_FILTER_HEADER_
#define _SUSAN_FILTER_HEADER_

// VPVE inclusions
#include <image_filter.h>

/**
 *  @ingroup filter_pack
 *  @brief This class is the SUSAN Filter
 *  @author Serge Catudal
 *  @version 1.0
 *  @date 05/03/2004
 */
class susan_filter : public image_filter{
 public:
  /// Constructor
  susan_filter();
  /// Copy constructor
  susan_filter(const susan_filter&);
  /// Overload the equal (=) operator
  susan_filter& operator=(const susan_filter&);
  /// Destructor
  ~susan_filter();

  // Access functions
  void set_bt(int b)   {bt = b;}
  void set_dt(float d) {dt = d;}

  // SUSAN filter functions
  int    median(int *p);
```

```
  int*  setup_brightness_lut(int thresh, int form);
  int** apply_filter(int **img_in);
  void  generate_matlab_file(string parameter_filename,string matlab_filename,
                             int** img_src, int** img_noisy, float noisy_quality);

 private:
  /// Brightness threshold (Brightness control)
  int   bt;
  /// Distance threshold (Spatial control)
  float dt;
};

#endif
```

D.2.2 Fichier susan_filter.cc

```
//-----------------------------------------------------------------------------
// Title          : SUSAN image filter
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : susan_filter.cc
// Author         : Serge CATUDAL
// Created        : 05/03/2004
// Last modified  : 29/04/2004
//-----------------------------------------------------------------------------
// Description    : Smallest Univalue Segment Assimilating Nucleus (SUSAN)
//-----------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 05/03/2004 : created
//-----------------------------------------------------------------------------

#include <susan_filter.h>

//-----------------------------------------------------------------------------
// Default Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
susan_filter::susan_filter() : image_filter(VPVE_SUSAN_FILTER){
  bt = 0;
  dt = 0;
}

//-----------------------------------------------------------------------------
// Copy Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
susan_filter::susan_filter(const susan_filter& sf) : image_filter(sf){
  bt = sf.bt;
  dt = sf.dt;
}

//-----------------------------------------------------------------------------
// = Operator Overloading
/**
 * @brief
 */
//-----------------------------------------------------------------------------
susan_filter& susan_filter::operator=(const susan_filter& sf){
  if(this != &sf){
    bt = sf.bt;
    dt = sf.dt;
  }
  return *this;
}

//-----------------------------------------------------------------------------
// Destructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
susan_filter::~susan_filter(){
}

//-----------------------------------------------------------------------------
```

```
// SUSAN Filter Functions
//-----------------------------------------------------------------------------
/**
 * @brief Evaluate the median value
 */
int susan_filter::median(int* p){
  int reg2_0,reg2_1,reg2_2,reg2_3,reg2_4,reg2_5,reg2_6,reg2_7;
  int reg3_0,reg3_1,reg3_2,reg3_3;
  int reg4_0,reg4_1;
  int result;

  // Stage 2
  if(p[0] > p[1]){
    reg2_0 = p[1];
    reg2_1 = p[0];
  } else {
    reg2_0 = p[0];
    reg2_1 = p[1];
  }

  if(p[2] > p[3]){
    reg2_2 = p[3];
    reg2_3 = p[2];
  } else {
    reg2_2 = p[2];
    reg2_3 = p[3];
  }

  if(p[4] > p[5]){
    reg2_4 = p[5];
    reg2_5 = p[4];
  } else {
    reg2_4 = p[4];
    reg2_5 = p[5];
  }

  if(p[6] > p[7]){
    reg2_6 = p[7];
    reg2_7 = p[6];
  } else {
    reg2_6 = p[6];
    reg2_7 = p[7];
  }

  // Stage 3
  if(reg2_0 > reg2_2)
    reg3_0 = reg2_0;
  else
    reg3_0 = reg2_2;

  if(reg2_1 > reg2_3)
    reg3_1 = reg2_3;
  else
    reg3_1 = reg2_1;

  if(reg2_4 > reg2_6)
    reg3_2 = reg2_4;
  else
    reg3_2 = reg2_6;

  if(reg2_5 > reg2_7)
    reg3_3 = reg2_7;
  else
    reg3_3 = reg2_5;

  // Stage 4
  if(reg3_1 > reg3_3)
    reg4_0 = reg3_1;
  else
    reg4_0 = reg3_3;

  if(reg3_0 > reg3_2)
    reg4_1 = reg3_2;
  else
    reg4_1 = reg3_0;

  // Stage 5
  result = (reg4_0 + reg4_1) / 2;

  // Free the memory
  delete[] p;
  p = NULL;

  return result;
}
```

```
/**
 * @brief Generate the Brightness Look-Up-Table (LUT)
 */
int* susan_filter::setup_brightness_lut(int thresh, int form){
  float temp;
  int   k;
  int   *b_lut;

  // Create LUT
  b_lut       = new int[513];
  for(k=-256; k<257; k++){
    temp            = ((float)k) / ((float)thresh);
    temp            = temp * temp;
    temp            = 100.0 * exp(-temp);
    b_lut[256 + k] = (int)temp;
  }

  return b_lut;
}

/**
 * @brief Compute the Smallest Univalue Segment Assimilating Nucleus (SUSAN) filter
 */
int** susan_filter::apply_filter(int **img_in){
  float temp;
  int   **img_out;
  int   **enlarge_img;
  int   area,total,tmp,brightness;
  int   j,i;
  int   y,x;
  int   *median_pixels;
  int   centre;          // Centre pixel of the gaussian mask
  int   *b_lut;          // Brightness LUT (Look-Up-Table)
  int   cp;              // Centre Pixel Brightness
  int   **g_mask;        // Gaussian mask
  int   mask_size;       // Gaussian mask size

  // Initialize img_out
  img_out = new int*[img_height];
  for(j=0; j<img_height; j++)
    img_out[j] = new int[img_width];

  // Create Brightness LUT
  b_lut = setup_brightness_lut(bt,2);

  // Create the gaussian mask
  temp      = -(dt*dt);
  mask_size = ((int)(1.5 * dt)) + 1;
  g_mask    = new int*[(2 * mask_size) + 1];
  for(j=-mask_size; j<=mask_size; j++){
    g_mask[j+mask_size] = new int[(mask_size*2) + 1];
    for(i=-mask_size; i<=mask_size; i++)
      g_mask[j+mask_size][i+mask_size] = (int)(100.0 * exp(((float)((i*i)+(j*j))) / temp));
  }

  // Enlarge img_in
  enlarge_img = new int*[img_height + (2 * mask_size)];
  for(j=0; j<img_width + (2 * mask_size); j++)
    enlarge_img[j] = new int[img_width + (2 * mask_size)];
  for(j=mask_size; j<(img_height + mask_size); j++)  // Copy img_in
    for(i=mask_size; i<(img_width + mask_size); i++)
      enlarge_img[j][i] = img_in[j - mask_size][i - mask_size];
  for(j=0; j<mask_size; j++)                         // Copy top and bottom rows
    for(i=mask_size; i<(img_width + mask_size); i++){
      enlarge_img[j][i] = enlarge_img[((2 * mask_size) + 1) - j][i];
      enlarge_img[j+img_height+mask_size][i] = enlarge_img[(img_height+mask_size-2)- j][i];
    }
  for(i=0; i<mask_size; i++)                         // Copy left and right columns
    for(j=0; j<(img_height + (2 * mask_size)); j++){
      enlarge_img[j][i] = enlarge_img[j][((2 * mask_size) + 1) - i];
      enlarge_img[j][i+img_width+mask_size] = enlarge_img[j][(img_width+ mask_size-2)-i];
    }

  // Filter the image
  for(j=mask_size; j<(img_height + mask_size); j++){
    for(i=mask_size; i<(img_width + mask_size); i++){
      area    = 0;
      total   = 0;
      centre = enlarge_img[j][i];
      cp      = 256 + centre;

      // Apply the gaussian mask
      for(y=-mask_size; y<= mask_size; y++){
        for(x=-mask_size; x<= mask_size; x++){
          brightness = enlarge_img[j+y][i+x];
          tmp        = g_mask[y + mask_size][x + mask_size] * b_lut[cp - brightness];
```

```
            area        = area + tmp;
            total       = total + (tmp * brightness);
          }
        }

        tmp = area - 10000;
        if(tmp == 0){
          // Generate the median vector
          median_pixels     = new int[8];
          median_pixels[0] = enlarge_img[j-1][i-1];
          median_pixels[1] = enlarge_img[j-1][i];
          median_pixels[2] = enlarge_img[j-1][i+1];
          median_pixels[3] = enlarge_img[j][i-1];
          median_pixels[4] = enlarge_img[j][i+1];
          median_pixels[5] = enlarge_img[j+1][i-1];
          median_pixels[6] = enlarge_img[j+1][i];
          median_pixels[7] = enlarge_img[j+1][i+1];

          // Compute the median value
          img_out[j - mask_size][i - mask_size] = median(median_pixels);
        } else {
          img_out[j - mask_size][i - mask_size] = ((total - (centre * 10000)) / tmp);
        }
      }
    }

    // Free the memory
    for(j=0; j<img_width + (2 * mask_size); j++)
      delete[] enlarge_img[j];
    for(j=0; j<(2 * mask_size) + 1; j++)
      delete[] g_mask[j];
    delete[] b_lut;
    delete[] enlarge_img;
    delete[] g_mask;
    b_lut        = NULL;
    enlarge_img = NULL;
    g_mask       = NULL;

    return img_out;
}

/**
 * @brief Generate the SUSAN matlab files
 */
void susan_filter::generate_matlab_file(string parameter_filename,string matlab_filename,
                                        int** img_src, int** img_noisy, float noisy_quality){
    char        *filename;
    float       d_t, dt_start, dt_end;
    float       img_quality;
    float       ratio;
    int         b_t, bt_start, bt_end;
    int         j;
    int         **img_filtered;
    string      tmp_filename;
    ssim_index *my_ssim_index;

    ifstream ReadTXT;
    ofstream WriteMatlab_results;

    // Parameter file content:
    //
    // Image Filename
    // Brightness Threshold start
    // Brightness Threshold end
    // Distance Threshold start
    // Distance Threshold end
    //
    // Example of the content of the parameter file:
    //
    // lena_256.pgm
    // 5
    // 55
    // 0.6
    // 4.5

    // Open parameter file
    ReadTXT.open(parameter_filename.c_str());

    // Save matlab results file
    filename = new char[40];
    strcat(filename,matlab_filename.c_str());
    strcat(filename,"_results.m");
    WriteMatlab_results.open(filename);
    delete[] filename;
    filename = NULL;
```

```
  if(!ReadTXT.fail() && !WriteMatlab_results.fail()){
    // Load the source image
    ReadTXT >> tmp_filename;

    // Load Brightness threshold parameters
    ReadTXT >> bt_start >> bt_end;
    cout << "\tBrightness_threshold_parameters_(Brightness_Control):_" << endl;
    cout << "\t\tStart:_" << bt_start << endl;
    cout << "\t\tEnd___:_" << bt_end << endl;

    // Load Distance threshold parameters
    ReadTXT >> dt_start >> dt_end;
    cout << "\tDistance_threshold_parameters_(Spatial_Control):_" << endl;
    cout << "\t\tStart:_" << dt_start << endl;
    cout << "\t\tEnd___:_" << dt_end << endl;
    cout <<"-------------------------------------------------------------------------------\n";

    // Initialize the ssim_index object
    my_ssim_index = new ssim_index();
    my_ssim_index->set_image_height(img_height);
    my_ssim_index->set_image_width(img_width);
    my_ssim_index->set_ssim_map();

    // Filter the image with different SUSAN parameters
    WriteMatlab_results << "susan_filter=[";
    for(b_t = bt_start; b_t <= bt_end; b_t++){
      for(d_t = dt_start; d_t <= dt_end; d_t = d_t + 0.1){
        set_bt(b_t);
        set_dt(d_t);
        img_filtered = apply_filter(img_noisy);
        img_quality  = (float)(my_ssim_index->compute_ssim_index(img_src,img_filtered));
        ratio        = noisy_quality / img_quality;
        if(d_t != dt_end)
          WriteMatlab_results << ratio << "_";
        else
          WriteMatlab_results << ratio;

        // Free the memory
        for(j=0; j<img_height; j++)
          delete[] img_filtered[j];
        delete[] img_filtered;
        img_filtered = NULL;
      }
      if(b_t != bt_end)
        WriteMatlab_results << ",\n";
    }
    WriteMatlab_results << "];";

    // Free the memory
    delete my_ssim_index;
  }

  ReadTXT.close();
  WriteMatlab_results.close();
}
```

D.3 Code source du filtre adaptatif Wiener

D.3.1 Fichier wiener_filter.h

```
//-----------------------------------------------------------------------------
// Title          : Model C/C++ du Filtre Wiener
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : Cmodel_wf.h
// Author         : Serge CATUDAL
// Created        : 04/12/2003
// Last modified  : 29/04/2004
//-----------------------------------------------------------------------------
// Description    : Model C/C++ du Filtre Wiener deriver du model SystemC
//-----------------------------------------------------------------------------
// Copyright (c) 2003 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 04/12/2003 : created
//-----------------------------------------------------------------------------
```

```
#ifndef _CMODEL_WF_HEADER_
#define _CMODEL_WF_HEADER_

// VPVE inclusions
#include <image_filter.h>

/**
 *  @ingroup filter_pack
 *  @brief This class is the Wiener Filter
 *  @author Serge Catudal
 *  @version 1.0
 *  @date 04/12/2003
 */
class cmodel_wf : public image_filter{

 public:
  /// Constructor
  cmodel_wf();
  /// Copy constructor
  cmodel_wf(const cmodel_wf&);
  /// Overload the equal (=) operator
  cmodel_wf& operator=(const cmodel_wf&);
  /// Destructor
  ~cmodel_wf();

  // Access functions

  // Local Mean and Local Variance module functions
  float  LM_div_by_9(float);
  float  LM_mask_3x3(float,float,float,bool);
  float* LV_3_mult(float,float,float);
  float  LV_div_by_9(float);
  float  LV_mask_3x3(float*,bool);
  float  LV_mult(float);
  float  LV_substraction(float,float);

  // Wiener Filter functions
  int** apply_filter(int **img_in);
  void  reset(void);
  void  generate_matlab_file(string parameter_filename,string matlab_filename,
                             int** img_src, int** img_noisy, float noisy_quality);

 private:
  /// Local Mean mask 3x3 cycle counter
  int   LM_mask_3x3_cycle_count;
  /// Local Mean mask 3x3 variable container pointer
  int   LM_mask_3x3_ptr;
  /// Local Mean mask 3x3 variable container
  float *LM_mask_3x3_var;
  /// Local Variance mask 3x3 cycle counter
  int   LV_mask_3x3_cycle_count;
  /// Local Variance mask 3x3 variable container pointer
  int   LV_mask_3x3_ptr;
  /// Local Variance mask 3x3 variable container
  float *LV_mask_3x3_var;
};

#endif
```

D.3.2 Fichier wiener_filter.cc

```
//-----------------------------------------------------------------------------
// Title          : Model C/C++ du Filtre Wiener
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : Cmodel_wf.cc
// Author         : Serge CATUDAL
// Created        : 05/12/2003
// Last modified  : 29/04/2004
//-----------------------------------------------------------------------------
// Description    : Model C/C++ du Filtre Wiener deriver du model SystemC
//-----------------------------------------------------------------------------
// Copyright (c) 2003 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 05/12/2003 : created
//-----------------------------------------------------------------------------

#include <Cmodel_wf.h>
```

```
//-----------------------------------------------------------------------------
// Default Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
cmodel_wf::cmodel_wf() : image_filter(VPVE_REGULAR_WIENER_FILTER){
  LM_mask_3x3_cycle_count = 0;
  LM_mask_3x3_ptr         = 0;
  LM_mask_3x3_var         = new float[3];
  LV_mask_3x3_cycle_count = 0;
  LV_mask_3x3_ptr         = 0;
  LV_mask_3x3_var         = new float[3];
  for(int k=0; k<3; k++){
    LM_mask_3x3_var[k] = 0;
    LV_mask_3x3_var[k] = 0;
  }
}

//-----------------------------------------------------------------------------
// Copy Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
cmodel_wf::cmodel_wf(const cmodel_wf& M) : image_filter(M){
  LM_mask_3x3_cycle_count = M.LM_mask_3x3_cycle_count;
  LM_mask_3x3_ptr         = M.LM_mask_3x3_ptr;
  LM_mask_3x3_var         = M.LM_mask_3x3_var;
  LV_mask_3x3_cycle_count = M.LV_mask_3x3_cycle_count;
  LV_mask_3x3_ptr         = M.LV_mask_3x3_ptr;
  LV_mask_3x3_var         = M.LV_mask_3x3_var;
}

//-----------------------------------------------------------------------------
// = Operator Overloading
/**
 * @brief
 */
//-----------------------------------------------------------------------------
cmodel_wf& cmodel_wf::operator=(const cmodel_wf& M){
  if(this != &M){
    LM_mask_3x3_cycle_count = M.LM_mask_3x3_cycle_count;
    LM_mask_3x3_ptr         = M.LM_mask_3x3_ptr;
    LM_mask_3x3_var         = M.LM_mask_3x3_var;
    LV_mask_3x3_cycle_count = M.LV_mask_3x3_cycle_count;
    LV_mask_3x3_ptr         = M.LV_mask_3x3_ptr;
    LV_mask_3x3_var         = M.LV_mask_3x3_var;
  }
  return *this;
}

//-----------------------------------------------------------------------------
// Destructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
cmodel_wf::~cmodel_wf(){
  delete[] LM_mask_3x3_var;
  delete[] LV_mask_3x3_var;
  LM_mask_3x3_var = NULL;
  LV_mask_3x3_var = NULL;
}

//-----------------------------------------------------------------------------
// Local Mean and Local Variance module Functions
//-----------------------------------------------------------------------------
/**
 * @brief Compute the module division by 9 from the Local Mean module
 */
float cmodel_wf::LM_div_by_9(float val_in){
  float result;

  result = val_in / 9;

  return result;
}

/**
 * @brief Compute the module mask 3x3 from the Local Mean module
 */
float cmodel_wf::LM_mask_3x3(float val_in_0,float val_in_1,float val_in_2,bool new_line){
  float result;
  float temp_result;
```

```
  temp_result = val_in_0 + val_in_1 + val_in_2;

  if(new_line){
    LM_mask_3x3_cycle_count = 0;
    LM_mask_3x3_ptr         = 0;
  }

  LM_mask_3x3_var[LM_mask_3x3_ptr] = temp_result;
  LM_mask_3x3_ptr++;
  if(LM_mask_3x3_ptr == 3)
    LM_mask_3x3_ptr = 0;

  if(LM_mask_3x3_cycle_count == 2){
    result = LM_mask_3x3_var[0] + LM_mask_3x3_var[1] + LM_mask_3x3_var[2];
    LM_mask_3x3_cycle_count++;
  } else if(LM_mask_3x3_cycle_count == 3){
    result = LM_mask_3x3_var[0] + LM_mask_3x3_var[1] + LM_mask_3x3_var[2];
  } else {
    LM_mask_3x3_cycle_count++;
  }

  return result;
}

/**
 * @brief Compute the module multiplication (3) from the Local Variance module
 */
float* cmodel_wf::LV_3_mult(float val_in_0,float val_in_1,float val_in_2){
  float *result = new float[3];
  float temp_result_0;
  float temp_result_1;
  float temp_result_2;

  temp_result_0 = val_in_0 * val_in_0;
  temp_result_1 = val_in_1 * val_in_1;
  temp_result_2 = val_in_2 * val_in_2;

  result[0] = temp_result_0;
  result[1] = temp_result_1;
  result[2] = temp_result_2;

  return result;
}

/**
 * @brief Compute the module division by 9 from the Local Variance module
 */
float cmodel_wf::LV_div_by_9(float val_in){
  float result;

  result = val_in / 9;

  return result;
}

/**
 * @brief Compute the module mask 3x3 from the Local Variance module
 */
float cmodel_wf::LV_mask_3x3(float* vector_in, bool new_line){
  float result;
  float temp_result;

  temp_result = vector_in[0] + vector_in[1] + vector_in[2];

  if(new_line){
    LV_mask_3x3_cycle_count = 0;
    LV_mask_3x3_ptr         = 0;
  }

  LV_mask_3x3_var[LV_mask_3x3_ptr] = temp_result;
  LV_mask_3x3_ptr++;
  if(LV_mask_3x3_ptr == 3)
    LV_mask_3x3_ptr = 0;

  if(LV_mask_3x3_cycle_count == 2){
    result = LV_mask_3x3_var[0] + LV_mask_3x3_var[1] + LV_mask_3x3_var[2];
    LV_mask_3x3_cycle_count++;
  } else if(LV_mask_3x3_cycle_count == 3){
    result = LV_mask_3x3_var[0] + LV_mask_3x3_var[1] + LV_mask_3x3_var[2];
  } else {
    LV_mask_3x3_cycle_count++;
  }

  // Free the memory
  delete[] vector_in;
  vector_in = NULL;
```

```
  return result;
}

/**
 * @brief Compute the module multiplication from the Local Variance module
 */
float cmodel_wf::LV_mult(float val_in){
  float result;

  result = val_in * val_in;

  return result;
}

/**
 * @brief Compute the module substraction from the Local Variance module
 */
float cmodel_wf::LV_substraction(float val_in_0, float val_in_1){
  float result;

  result = val_in_1 - val_in_0;

  return result;
}

//-----------------------------------------------------------------------------
// Wiener Filter Functions
//-----------------------------------------------------------------------------
/**
 * @brief Compute the Wiener Filter algorithm
 */
int** cmodel_wf::apply_filter(int** img_in){
  int    i,j;
  int    x,y;
  int    val_0,val_1,val_2;
  bool   new_line;
  float lm_temp_result, lv_temp_result;
  int    **img_out;
  float **local_mean;
  float **local_variance;
  float noise_estimation;
  float ne_reg1_0,ne_reg1_1,ne_reg1_3;
  float ne_reg2_0,ne_reg2_1;
  float ne_reg3_0;

  // Variable initialization
  x              = 0;
  y              = 0;
  img_out        = new int*[img_height];
  local_mean     = new float*[img_height];
  local_variance = new float*[img_height];
  for(j = 0; j < img_height; j++){
    img_out[j]        = new int[img_width];
    local_mean[j]     = new float[img_width];
    local_variance[j] = new float[img_width];
  }
  noise_estimation = 0;
  reset();

  // Compute local mean and local variance
  for(j = 0; j < img_height; j++){
    for(i = -1; i < (img_width + 1); i++){
      // Assign values for the vertical vector
      if(i == -1 || i == img_width){
        val_0 = 0;
        val_1 = 0;
        val_2 = 0;
        if(i == -1)
          new_line = true;
      } else {
        if(j == 0){
          val_0 = 0;
          val_1 = img_in[j][i];
          val_2 = img_in[j + 1][i];
        } else if (j == (img_height - 1)){
          val_0 = img_in[j - 1][i];
          val_1 = img_in[j][i];
          val_2 = 0;
        } else {
          val_0 = img_in[j - 1][i];
          val_1 = img_in[j][i];
          val_2 = img_in[j + 1][i];
        }
        new_line = false;
      }
```

```
      // Compute local_mean
      lm_temp_result = LM_mask_3x3(val_0,val_1,val_2,new_line);
      if(LM_mask_3x3_cycle_count == 3){
        local_mean[y][x] = LM_div_by_9(lm_temp_result);
      }

      // Compute local variance
      lv_temp_result = LV_mask_3x3(LV_3_mult(val_0,val_1,val_2),new_line);
      if(LV_mask_3x3_cycle_count == 3){
        local_variance[y][x] = LV_substraction(LV_mult(local_mean[y][x]),
                                               LV_div_by_9(lv_temp_result));
        noise_estimation = noise_estimation + local_variance[y][x];
      }

      // update table pointer
      if(LV_mask_3x3_cycle_count == 3){
        x++;
        if(x == img_width){
          x = 0;
          y++;
        }
      }
    }
  }

  // Compute noise estimation
  noise_estimation = noise_estimation / (img_height * img_width);

  // Filter the image
  for(j = 0; j < img_height; j++){
    for(i = 0; i < img_width; i++){
      // Stage 2
      if(local_variance[j][i] < noise_estimation)
        ne_reg1_0 = noise_estimation;
      else
        ne_reg1_0 = local_variance[j][i];
      ne_reg1_1 = local_variance[j][i] - noise_estimation;
      ne_reg1_3 = img_in[j][i] - local_mean[j][i];

      // Stage 3
      ne_reg2_0 = ne_reg1_3 / ne_reg1_0;
      if(ne_reg1_1 < 0)
        ne_reg2_1 = 0;
      else
        ne_reg2_1 = ne_reg1_1;

      // Stage 4
      ne_reg3_0 = ne_reg2_0 * ne_reg2_1;

      // Stage 5
      img_out[j][i] = (int)(ne_reg3_0 + local_mean[j][i]);
    }
  }

  // Free the memory
  for(j=0; j<img_height; j++){
    delete[] local_mean[j];
    delete[] local_variance[j];
  }
  delete[] local_mean;
  delete[] local_variance;
  local_mean     = NULL;
  local_variance = NULL;

  return img_out;
}

/**
 * @brief Reset the C/C++ Wiener Filter model
 */
void cmodel_wf::reset(void){
  LM_mask_3x3_cycle_count = 0;
  LM_mask_3x3_ptr         = 0;
  LV_mask_3x3_cycle_count = 0;
  LV_mask_3x3_ptr         = 0;
  for(int k=0; k<3; k++){
    LM_mask_3x3_var[k] = 0;
    LV_mask_3x3_var[k] = 0;
  }
}

/**
 * @brief Generate the regular Wiener filter matlab files
 */
void cmodel_wf::generate_matlab_file(string parameter_filename,string matlab_filename,
```

```
                                        int** img_src, int** img_noisy, float noisy_quality){
  cout << "\t\tFunction not implemented -- Regular Wiener Filter " << endl;
  cout << "----------------------------------------------------------------------\n";
}
```

ANNEXE E

FICHIERS DE SIMULATIONS

E.1 Code source pour simuler les filtres numériques

E.1.1 Fichier Ximage.c

```
//-----------------------------------------------------------------------------
// Title          : Modele C/C++ du Filtre Numerique
// Project        : Video Processing Module (VPM)
//-----------------------------------------------------------------------------
// File           : numerical_filter.c
// Author         : Serge CATUDAL
// Created        : 11/12/2003
// Last modified  : 14/12/2003
//-----------------------------------------------------------------------------
// Description    :
//-----------------------------------------------------------------------------
// Copyright (c) 2003 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 11/12/2003 : created
//-----------------------------------------------------------------------------

// Architecture optimization definition

// Constant definitions

// C/C++ inclusions
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>

// SynapC inclusions
#include <gFix.h>
#include <Xpublic.h>

// Global external variable declaration
extern int LAST_CHECK;
extern int SARGE_FLAG0;

// Global variable declaration

// Numerical Filter Functions
int** compute_nf(int** img,int height,int width);
void  put_image(int img_num,int **img_src,int **img_filtered,
                float img_quality,int img_height,int img_width);

void XSmain(){
  int    **img_src;
  int    **img_noisy;
  int    **img_filtered;
  int    img_height,img_width;
  int    img_num = 0;
  int    row, col;
  int    the_int;
  float  img_quality;
  float  the_float;
  float  the_height,the_width;

  while((getinput(&img_quality) == 1) && (getinput(&the_height) == 1) &&
        (getinput(&the_width) == 1)){
    // Get image size
```

```
    img_height = (int)the_height;
    img_width  = (int)the_width;

    // Initialize the original and noisy image
    img_src   = new int*[img_height];
    img_noisy = new int*[img_height];
    for(row=0; row<img_height; row++){
      img_src[row]   = new int[img_width];
      img_noisy[row] = new int[img_width];
    }

    // Get the original image
    for(row=0; row<img_height; row++){
      for(col=0; col<img_width; col++){
        if(getinput(&the_float) == 1)
          img_src[row][col] = (int)the_float;
        else
          fprintf(stderr,"Original image is wrong size.\n");
      }
    }

    // Get the noisy image
    for(row=0; row<img_height; row++){
      for(col=0; col<img_width; col++){
        if(getinput(&the_float) == 1)
          img_noisy[row][col] = (int)the_float;
        else
          fprintf(stderr,"Noisy image is wrong size.\n");
      }
    }

    // Execute the image filter
    img_filtered = compute_nf(img_noisy,img_height,img_width);

    // Save the result
    put_image(img_num,img_src,img_filtered,img_quality,img_height,img_width);
    img_num++;

    // Free the memory
    for(row=0; row<img_height; row++){
      delete[] img_src[row];
      delete[] img_noisy[row];
      delete[] img_filtered[row];
    }
    delete [] img_src;
    delete [] img_noisy;
    delete [] img_filtered;
    img_src      = NULL;
    img_noisy    = NULL;
    img_filtered = NULL;

    // WARNING
    // Encountered a negative pixel in the fixed filter result
    if(SARGE_FLAG0 == 1)
      break;
  }
}

//-------------------------------------------------------------------------
// Numerical Filter Functions
//-------------------------------------------------------------------------
// Compute the Numerical Filter algorithm
int** compute_nf(int** img,int height,int width){

  // Numerical Filter Algorithm code here

  return img_filtered;
}

// Save the data for the Xerror algorithm
void put_image(int img_num,int **img_src,int **img_filtered,
               float img_quality,int img_height,int img_width){
  int  row,col;
  int  itr;
  char *ctmp          = new char[10];
  char *filename      = new char[30];
  char *filename_err  = new char[30];
  char *header_tmp    = new char[20];

  ofstream WriteBIN;
  ofstream WriteTXT;

  // Save the image quality
  putoutput(img_quality);

  // Save the image size
```

```
  putoutput((float)img_height);
  putoutput((float)img_width);
  if(LAST_CHECK == 1){
    // Create the error filename (when the program encounter a negative pixel)
    strcat(filename_err,"neg_pixel_report_img_");
    sprintf(ctmp,"%d",img_num);
    strcat(filename_err,ctmp);
    strcat(filename_err,".log");
    WriteTXT.open(filename_err);

    // Create raw PGM image file header
    strcat(header_tmp,"P5\n");
    sprintf(ctmp,"%d",img_height);
    strcat(header_tmp,ctmp);
    strcat(header_tmp," ");
    sprintf(ctmp,"%d",img_width);
    strcat(header_tmp,ctmp);
    strcat(header_tmp,"\n255\n");

    // Create filename
    strcat(filename,"img_filter_fixed_");
    sprintf(ctmp,"%d",img_num);
    strcat(filename,ctmp);
    strcat(filename,".pgm");

    // Write raw PGM image file
    WriteBIN.open(filename, ios::binary);
    WriteBIN.seekp(0,ios::beg);
    // Write header
    for(itr=0; itr<strlen(header_tmp); itr++)
      WriteBIN.write((char *) &header_tmp[itr],sizeof(char));
  }

  // Save the original and the filtered image (interlaced pixel)
  for(row=0; row<img_height; row++){
    for(col=0; col<img_width; col++){
      putoutput((float)img_src[row][col]);
      putoutput((float)img_filtered[row][col]);
      // Write each pixels into the raw PGM file
      if(LAST_CHECK == 1){
        WriteBIN.write((char *) &(unsigned char)img_filtered[row][col],sizeof(unsigned char));
        if(img_filtered[row][col] < 0)
          WriteTXT << "Negative_Pixel[" << row << "][" << col << "]_=_"
                   << img_filtered[row][col] << endl;
      }
    }
  }
  if(LAST_CHECK == 1){
    WriteBIN.close();
    WriteTXT.close();
  }

  // Free the memory
  delete[] ctmp;
  delete[] filename;
  delete[] filename_err;
  delete[] header_tmp;
  ctmp         = NULL;
  filename     = NULL;
  filename_err = NULL;
  header_tmp   = NULL;
}
```

E.1.2 Fichier Ximage_error.c

```
//-------------------------------------------------------------------------
// Title          : Structural SIMilarity Index (SSIM Index)
// Project        : Video Processing Module
//-------------------------------------------------------------------------
// File           : Ximage_error.c
// Author         : Serge Catudal
// Created        : 12/12/2003
// Last modified  : 12/12/2003
//-------------------------------------------------------------------------
// Description    : Objective Image Quality Metric
//-------------------------------------------------------------------------
// Copyright (c) 2003 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-------------------------------------------------------------------------
// Modification history :
// 12/12/2003 : created
```

```
//-------------------------------------------------------------------------------

// C/C++ inclusions
#include <stdio.h>
#include <stdlib.h>

// SynapC inclusions
#include <Xpublic.h>

// Size of the circular symmetric gaussian weighting function
#define size_window 11

// Constants in the SSIM index formula
#define K1 0.01
#define K2 0.03
#define L   255

// Global external variable declaration
extern int   SARGE_FLAG0;  // Negative pixel detector (1 = negative pixel)
extern float SARGE_FLAG1;  // IRATIO objective
extern float SARGE_FLAG2;  // MAX    objective

// Circular symmetric gaussian weighting function
const double window[size_window][size_window] = {
  {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000},
  {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
  {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
  {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
  {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
  {0.0003,0.0020,0.0096,0.0291,0.0567,0.0708,0.0567,0.0291,0.0096,0.0020,0.0003},
  {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
  {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
  {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
  {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
  {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000}
};

void Xerror(){
  bool   first_time = true;
  double C1,C2;
  double denominator1,denominator2;
  double numerator1,numerator2;
  int    border;
  int    i,j;
  int    m,n;
  int    current_threshold,max_threshold;
  int    **img1_img2,**img1_sq,**img2_sq;
  double **mu1_mu2,**mu1,**mu2;
  double **mu1_sq,**mu2_sq;
  double **sigma12,**sigma1_sq,**sigma2_sq;
  double result = 0.0;
  double ssim_map;
  int    **original,**distorted;
  int    height,width;
  float  filtered_floating;
  float  floating,fixed,quality,tmp;
  float  the_height,the_width;
  float  ratio,smallest_ratio;

  // Variable initialization
  border          = ((size_window - 1) / 2);
  smallest_ratio = 1.0;

  // Constants initialiazation
  C1 = (K1*L)*(K1*L);
  C2 = (K2*L)*(K2*L);

  // MAX threshold initialization
  max_threshold = 0;

  while((getoutput(&floating,&quality) == 1) && (getoutput(&floating,&the_height) == 1)
        && (getoutput(&floating,&the_width) == 1)){
    // WARNING
    // Encountered a negative pixel in the fixed filter result
    if(SARGE_FLAG0 == 1)
      break;

    // Get image characteristics
    height = (int)the_height;
    width  = (int)the_width;

    // Initialize result
    result = 0.0;

    // Calculate the square for the original and distorted image
    original  = new int*[height];
```

```
distorted = new int*[height];
img1_img2 = new int*[height];
img1_sq   = new int*[height];
img2_sq   = new int*[height];
for(j = 0; j < height; j++){
  original[j]  = new int[width];
  distorted[j] = new int[width];
  img1_img2[j] = new int[width];
  img1_sq[j]   = new int[width];
  img2_sq[j]   = new int[width];
  for(i = 0; i < width; i++){
    if(getoutput(&tmp,&floating)!=1 || getoutput(&filtered_floating,&fixed)!=1){
      fprintf(stderr,"\n\t**_Error_**\n\n\t\tThe_size_of_the_image_is_incorrect\n");
      exit(1);
    }
    original[j][i]  = (int)floating;
    distorted[j][i] = (int)fixed;
    img1_img2[j][i] = original[j][i] * distorted[j][i];
    img1_sq[j][i]   = original[j][i] * original[j][i];
    img2_sq[j][i]   = distorted[j][i] * distorted[j][i];

    // Calculate MAX value
    current_threshold = abs(((int)filtered_floating) - distorted[j][i]);
    if(current_threshold > max_threshold)
      max_threshold = current_threshold;
  }
}

// Apply the 2D Filter on the mean intensity and the variance
mu1       = new double*[(height - size_window) + 1];
mu2       = new double*[(height - size_window) + 1];
sigma12   = new double*[(height - size_window) + 1];
sigma1_sq = new double*[(height - size_window) + 1];
sigma2_sq = new double*[(height - size_window) + 1];
mu1_mu2   = new double*[(height - size_window) + 1];
mu1_sq    = new double*[(height - size_window) + 1];
mu2_sq    = new double*[(height - size_window) + 1];
for(j = (0 + border); j < (height - border); j++){
  mu1[j-border]       = new double[(width - size_window) + 1];
  mu2[j-border]       = new double[(width - size_window) + 1];
  sigma12[j-border]   = new double[(width - size_window) + 1];
  sigma1_sq[j-border] = new double[(width - size_window) + 1];
  sigma2_sq[j-border] = new double[(width - size_window) + 1];
  mu1_mu2[j-border]   = new double[(width - size_window) + 1];
  mu1_sq[j-border]    = new double[(width - size_window) + 1];
  mu2_sq[j-border]    = new double[(width - size_window) + 1];
  for(i = (0 + border); i < (width - border); i++){
    mu1[j-border][i-border]       = 0;
    mu2[j-border][i-border]       = 0;
    sigma12[j-border][i-border]   = 0;
    sigma1_sq[j-border][i-border] = 0;
    sigma2_sq[j-border][i-border] = 0;
    // Apply the circular symmetric gaussian weighting function on each pixel
    for(n = (0 - border); n < (size_window - border); n++){
      for(m = (0 - border); m < (size_window - border); m++){
        mu1[j-border][i-border] =
          mu1[j-border][i-border] +
          ((double)(original[j+n][i+m]) * window[n+border][m+border]);
        mu2[j-border][i-border] =
          mu2[j-border][i-border] +
          ((double)(distorted[j+n][i+m]) * window[n+border][m+border]);
        sigma12[j-border][i-border] =
          sigma12[j-border][i-border] +
          ((double)(img1_img2[j+n][i+m]) * window[n+border][m+border]);
        sigma1_sq[j-border][i-border] =
          sigma1_sq[j-border][i-border] +
          ((double)(img1_sq[j+n][i+m]) * window[n+border][m+border]);
        sigma2_sq[j-border][i-border] =
          sigma2_sq[j-border][i-border] +
          ((double)(img2_sq[j+n][i+m]) * window[n+border][m+border]);
      }
    }

    // SSIM(x,y) = [l(x,y)] * [c(x,y)] * [s(x,y)]
    //
    //     where
    //
    //           l(x,y) is the luminance comparison
    //           l(x,y) = (2*mu_x*mu_y + C1) / (mu_x^2 + mu_y^2 + C1)
    //
    //           c(x,y) is the contrast comparison
    //           c(x,y) = (2*sigma_x*sigma_y + C2) / (sigma_x^2 + sigma_y^2 + C2)
    //
    //           s(x,y) is the structure comparison
    //           s(x,y) = (sigma_xy + C3) / (sigma_x*sigma_y + C3)
    //
```

```
      // SSIM(x,y) = [(2*mu_x*mu_y + C1) * (2*sigma_x*sigma_y + C2)] /
      //             [(mu_x^2 + mu_y^2 + C1) * (sigma_x^2 + sigma_y^2 + C2)]
      //
      mu1_mu2[j-border][i-border] = mu1[j-border][i-border] * mu2[j-border][i-border];
      mu1_sq[j-border][i-border] = mu1[j-border][i-border] * mu1[j-border][i-border];
      mu2_sq[j-border][i-border] = mu2[j-border][i-border] * mu2[j-border][i-border];
      sigma12[j-border][i-border] = (sigma12[j-border][i-border] -
                                     mu1_mu2[j-border][i-border]);
      sigma1_sq[j-border][i-border] = (sigma1_sq[j-border][i-border] -
                                       mu1_sq[j-border][i-border]);
      sigma2_sq[j-border][i-border] = (sigma2_sq[j-border][i-border] -
                                       mu2_sq[j-border][i-border]);
      numerator1 = ((2 * mu1_mu2[j-border][i-border]) + C1);
      numerator2 = ((2 * sigma12[j-border][i-border]) + C2);
      denominator1 = (mu1_sq[j-border][i-border] + mu2_sq[j-border][i-border] + C1);
      denominator2 = (sigma1_sq[j-border][i-border] + sigma2_sq[j-border][i-border] + C2);
      ssim_map = (numerator1 * numerator2) / (denominator1 * denominator2);
      result = result + (float)ssim_map;
    }
  }

  // A mean SSIM index (MSSIM) is used to evaluate the overall image quality
  //
  // MSSIM(X,Y) = (1/M)SUM(SSIM(x,y))
  //
  result = result / (((height - size_window) + 1) * ((width - size_window) + 1));

  // Calculate the ratio
  ratio = ((float)result) / quality;
  if(first_time){
    smallest_ratio = ratio;
    first_time     = false;
  } else {
    if(ratio < smallest_ratio)
      smallest_ratio = ratio;
  }

  // Free the memory
  for(j = 0; j < height; j++){
    delete[] original[j];
    delete[] distorted[j];
    delete[] img1_img2[j];
    delete[] img1_sq[j];
    delete[] img2_sq[j];
    if((j > (0 + border - 1)) && (j < (height - border))){
      delete[] mu1_mu2[j-border];
      delete[] mu1[j-border];
      delete[] mu2[j-border];
      delete[] mu1_sq[j-border];
      delete[] mu2_sq[j-border];
      delete[] sigma12[j-border];
      delete[] sigma1_sq[j-border];
      delete[] sigma2_sq[j-border];
    }
  }
  delete []img1_img2;
  delete []img1_sq;
  delete []img2_sq;
  delete []mu1_mu2;
  delete []mu1;
  delete []mu2;
  delete []mu1_sq;
  delete []mu2_sq;
  delete []sigma12;
  delete []sigma1_sq;
  delete []sigma2_sq;
  delete []original;
  delete []distorted;
  img1_img2 = NULL;
  img1_sq   = NULL;
  img2_sq   = NULL;
  mu1_mu2   = NULL;
  mu1       = NULL;
  mu2       = NULL;
  mu1_sq    = NULL;
  mu2_sq    = NULL;
  sigma12   = NULL;
  sigma1_sq = NULL;
  sigma2_sq = NULL;
  original  = NULL;
  distorted = NULL;

  // WARNING
  // One of the objectives is not met, jump to the next
  // optimization to save simulation time
  if(((1 - smallest_ratio) > SARGE_FLAG1) || (max_threshold > SARGE_FLAG2))
```

```
      break;
  }

  // WARNING
  // Encountered a negative pixel in the fixed filter result
  if(SARGE_FLAG0 == 1){
    smallest_ratio = 0;
    SARGE_FLAG0    = 0;
  }

  puterror("IRATIO",1-smallest_ratio);
  puterror("MAX",max_threshold);
}
```

E.2 Code source pour simuler les filtres numériques avec paramètres

E.2.1 Fichier Xparam.c

```
//-----------------------------------------------------------------------------
// Title          : Numerical image filter with parameters
// Project        : Video Processing Validation Environment
//-----------------------------------------------------------------------------
// File           : Xparam.c
// Author         : Serge CATUDAL
// Created        : 18/05/2004
// Last modified  : 02/06/2004
//-----------------------------------------------------------------------------
// Description    :
//-----------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 18/05/2004 : created
//-----------------------------------------------------------------------------

// Parameter optimization definition

// Architecture optimization definition

// Constant definitions

// C/C++ inclusions
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>

// SynapC inclusions
#include <gFix.h>
#include <Xpublic.h>

// Global external variable declaration
extern int LAST_CHECK;

// Hybrid Filter function
int** compute_param_nf(int** img_in, int img_height, int img_width);

// SynapC function
void  put_image(int img_num,int **img_src,int **img_filtered,
                float noisy_quality,int img_height,int img_width);

// SynapC main function
void XSmain(){
  int    **img_src;
  int    **img_noisy;
  int    **img_filtered;
  int    img_height,img_width;
  int    img_num = 0;
  int    row, col;
  int    the_int;
  float  the_float;
  float  the_height,the_width;
  float  the_noisy_quality;

  while((getinput(&the_noisy_quality) == 1) && (getinput(&the_height) == 1) &&
```

```
        (getinput(&the_width) == 1)){
    // Get image size
    img_height = (int)the_height;
    img_width  = (int)the_width;

    // Initialize the original and noisy image
    img_src   = new int*[img_height];
    img_noisy = new int*[img_height];
    for(row=0; row<img_height; row++){
      img_src[row]   = new int[img_width];
      img_noisy[row] = new int[img_width];
    }

    // Get the original image
    for(row=0; row<img_height; row++){
      for(col=0; col<img_width; col++){
        if(getinput(&the_float) == 1)
          img_src[row][col] = (int)the_float;
        else
          fprintf(stderr,"Original image is wrong size.\n");
      }
    }

    // Get the noisy image
    for(row=0; row<img_height; row++){
      for(col=0; col<img_width; col++){
        if(getinput(&the_float) == 1)
          img_noisy[row][col] = (int)the_float;
        else
          fprintf(stderr,"Noisy image is wrong size.\n");
      }
    }

    // Execute the image filter
    img_filtered = compute_param_nf(img_noisy,img_height,img_width);

    // Save the result
    put_image(img_num,img_src,img_filtered,the_noisy_quality,img_height,img_width);
    img_num++;

    // Free the memory
    for(row=0; row<img_height; row++){
      delete[] img_src[row];
      delete[] img_noisy[row];
      delete[] img_filtered[row];
    }
    delete []img_src;
    delete []img_noisy;
    delete []img_filtered;
    img_src      = NULL;
    img_noisy    = NULL;
    img_filtered = NULL;
  }
}

//-------------------------------------------------------------------------
// Numerical Filter with Parameters Functions
//-------------------------------------------------------------------------
// Comput the Hybrid filter
int** compute_param_nf(int** img_in, int img_height, int img_width){

  // Numerical Filter Algorithm with parameters code here

  return img_out;
}

//-------------------------------------------------------------------------
// SynapC Function
//-------------------------------------------------------------------------
// Save the data for the Xerror algorithm
void put_image(int img_num,int **img_src,int **img_filtered,
               float noisy_quality,int img_height,int img_width){
  int  row,col;
  int  itr;
  char *ctmp          = new char[10];
  char *filename      = new char[30];
  char *header_tmp    = new char[20];

  ofstream WriteBIN;

  // Save the noisy quality
  putoutput(noisy_quality);

  // Save the image size
  putoutput((float)img_height);
  putoutput((float)img_width);
```

```
  if(LAST_CHECK == 1){
    // Create raw PGM image file header
    strcat(header_tmp,"P5\n");
    sprintf(ctmp,"%d",img_height);
    strcat(header_tmp,ctmp);
    strcat(header_tmp," ");
    sprintf(ctmp,"%d",img_width);
    strcat(header_tmp,ctmp);
    strcat(header_tmp,"\n255\n");

    // Create filename
    strcat(filename,"img_filter_fixed_");
    sprintf(ctmp,"%d",img_num);
    strcat(filename,ctmp);
    strcat(filename,".pgm");

    // Write raw PGM image file
    WriteBIN.open(filename, ios::binary);
    WriteBIN.seekp(0,ios::beg);
    // Write header
    for(itr=0; itr<strlen(header_tmp); itr++)
      WriteBIN.write((char *) &header_tmp[itr],sizeof(char));
  }

  // Save the original and the filtered image (interlaced pixel)
  for(row=0; row<img_height; row++){
    for(col=0; col<img_width; col++){
      putoutput((float)img_src[row][col]);
      putoutput((float)img_filtered[row][col]);
      // Write each pixels into the raw PGM file
      if(LAST_CHECK == 1)
        WriteBIN.write((char *) &(unsigned char)img_filtered[row][col],
                       sizeof(unsigned char));
    }
  }
  if(LAST_CHECK == 1)
    WriteBIN.close();

  // Free the memory
  delete[] ctmp;
  delete[] filename;
  delete[] header_tmp;
  ctmp       = NULL;
  filename   = NULL;
  header_tmp = NULL;
}
```

E.2.2 Fichier Xparam_error.c

```
//-------------------------------------------------------------------------
// Title          : Structural SIMilarity Index (SSIM Index)
// Project        : Video Processing Validation Envirronment (VPVE)
//-------------------------------------------------------------------------
// File           : Xparam_error.c
// Author         : Serge Catudal
// Created        : 18/05/2004
// Last modified  : 26/05/2004
//-------------------------------------------------------------------------
// Description    : Objective Image Quality Metric
//-------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-------------------------------------------------------------------------
// Modification history :
// 18/05/2004 : created
//-------------------------------------------------------------------------

// C/C++ inclusions
#include <stdio.h>
#include <stdlib.h>

// SynapC inclusions
#include <Xpublic.h>

// Size of image database
#define size_img_db  5

// Size of the circular symmetric gaussian weighting function
#define size_window 11

// Constants in the SSIM index formula
```

```
#define K1 0.01
#define K2 0.03
#define L   255

// Global internal variable declaration
float floating_MSSIM_array[size_img_db];

// Global external variable declaration
extern int PAR_OPT;       // 1 = Parameter estimation, 0 = Parameter estimation + operands
extern int SARGE_FLAG0;   // Negative pixel detector (1 = negative pixel)

// Circular symmetric gaussian weighting function
const double window[size_window][size_window] = {
  {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000},
  {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
  {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
  {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
  {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
  {0.0003,0.0020,0.0096,0.0291,0.0567,0.0708,0.0567,0.0291,0.0096,0.0020,0.0003},
  {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
  {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
  {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
  {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
  {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000}
};

// SynapC error function
void Xerror(){
  int     border;
  double C1,C2;
  double denominator1,denominator2;
  double luminance, mean_luminance;
  double numerator1,numerator2;
  bool    first_time = true;
  int     i,j;
  int     m,n;
  int     current_threshold,max_threshold;
  int     **img1_img2,**img1_sq,**img2_sq;
  double **mu1_mu2,**mu1,**mu2;
  double **mu1_sq,**mu2_sq;
  double **sigma12,**sigma1_sq,**sigma2_sq;
  double result = 0.0;
  double ssim_map;
  int     **original,**distorted;
  int     height,width;
  float   filtered_floating;
  float   floating,fixed,quality,tmp;
  float   the_height,the_width;
  float   the_noisy_quality;
  float   ratio, smallest_ratio, sum_ratio, worst_ratio;
  float   worst_luminance;
  int     img_ptr;

  // Variable initialization
  border           = ((size_window - 1) / 2);
  img_ptr          = 0;
  mean_luminance   = 0.0;
  smallest_ratio   = 1.0;
  sum_ratio        = 0.0;
  worst_luminance = 1.0;

  // Constants initialiazation
  C1 = (K1*L)*(K1*L);
  C2 = (K2*L)*(K2*L);

  // MAX threshold initialization
  max_threshold = 0;

  while((getoutput(&floating,&the_noisy_quality) == 1) &&
        (getoutput(&floating,&the_height) == 1) && (getoutput(&floating,&the_width) == 1)){
    // WARNING
    // Encountered a negative pixel in the fixed filter result
    if(SARGE_FLAG0 == 1)
      break;

    // Get image characteristics
    height = (int)the_height;
    width  = (int)the_width;

    // Initialize result and mean luminance
    result          = 0.0;
    mean_luminance = 0.0;

    // Calculate the square for the original and distorted image
    original  = new int*[height];
    distorted = new int*[height];
```

```
img1_img2 = new int*[height];
img1_sq   = new int*[height];
img2_sq   = new int*[height];
for(j = 0; j < height; j++){
  original[j]  = new int[width];
  distorted[j] = new int[width];
  img1_img2[j] = new int[width];
  img1_sq[j]   = new int[width];
  img2_sq[j]   = new int[width];
  for(i = 0; i < width; i++){
    if(getoutput(&tmp,&floating)!=1 || getoutput(&filtered_floating,&fixed)!=1){
      fprintf(stderr,"\n\t**Error**\n\n\t\tThe size of the image is incorrect\n");
      exit(1);
    }
    original[j][i]  = (int)floating;
    distorted[j][i] = (int)fixed;
    img1_img2[j][i] = original[j][i] * distorted[j][i];
    img1_sq[j][i]   = original[j][i] * original[j][i];
    img2_sq[j][i]   = distorted[j][i] * distorted[j][i];

    if(PAR_OPT == 0){
      // Calculate MAX value
      current_threshold = abs(((int)filtered_floating) - distorted[j][i]);
      if(current_threshold > max_threshold)
        max_threshold = current_threshold;
    }
  }
}

// Apply the 2D Filter on the mean intensity and the variance
mu1       = new double*[(height - size_window) + 1];
mu2       = new double*[(height - size_window) + 1];
sigma12   = new double*[(height - size_window) + 1];
sigma1_sq = new double*[(height - size_window) + 1];
sigma2_sq = new double*[(height - size_window) + 1];
mu1_mu2   = new double*[(height - size_window) + 1];
mu1_sq    = new double*[(height - size_window) + 1];
mu2_sq    = new double*[(height - size_window) + 1];
for(j = (0 + border); j < (height - border); j++){
  mu1[j-border]       = new double[(width - size_window) + 1];
  mu2[j-border]       = new double[(width - size_window) + 1];
  sigma12[j-border]   = new double[(width - size_window) + 1];
  sigma1_sq[j-border] = new double[(width - size_window) + 1];
  sigma2_sq[j-border] = new double[(width - size_window) + 1];
  mu1_mu2[j-border]   = new double[(width - size_window) + 1];
  mu1_sq[j-border]    = new double[(width - size_window) + 1];
  mu2_sq[j-border]    = new double[(width - size_window) + 1];
  for(i = (0 + border); i < (width - border); i++){
    mu1[j-border][i-border]       = 0;
    mu2[j-border][i-border]       = 0;
    sigma12[j-border][i-border]   = 0;
    sigma1_sq[j-border][i-border] = 0;
    sigma2_sq[j-border][i-border] = 0;
    // Apply the circular symmetric gaussian weighting function on each pixel
    for(n = (0 - border); n < (size_window - border); n++){
      for(m = (0 - border); m < (size_window - border); m++){
        mu1[j-border][i-border] =
          mu1[j-border][i-border] +
          ((double)(original[j+n][i+m]) * window[n+border][m+border]);
        mu2[j-border][i-border] =
          mu2[j-border][i-border] +
          ((double)(distorted[j+n][i+m]) * window[n+border][m+border]);
        sigma12[j-border][i-border] =
          sigma12[j-border][i-border] +
          ((double)(img1_img2[j+n][i+m]) * window[n+border][m+border]);
        sigma1_sq[j-border][i-border] =
          sigma1_sq[j-border][i-border] +
          ((double)(img1_sq[j+n][i+m]) * window[n+border][m+border]);
        sigma2_sq[j-border][i-border] =
          sigma2_sq[j-border][i-border] +
          ((double)(img2_sq[j+n][i+m]) * window[n+border][m+border]);
      }
    }

    // SSIM(x,y) = [l(x,y)] * [c(x,y)] * [s(x,y)]
    //
    //    where
    //
    //          l(x,y) is the luminance comparison
    //          l(x,y) = (2*mu_x*mu_y + C1) / (mu_x^2 + mu_y^2 + C1)
    //
    //          c(x,y) is the contrast comparison
    //          c(x,y) = (2*sigma_x*sigma_y + C2) / (sigma_x^2 + sigma_y^2 + C2)
    //
    //          s(x,y) is the structure comparison
    //          s(x,y) = (sigma_xy + C3) / (sigma_x*sigma_y + C3)
```

```
      //
      // SSIM(x,y) = [(2*mu_x*mu_y + C1) * (2*sigma_x*sigma_y + C2)] /
      //             [(mu_x^2 + mu_y^2 + C1) * (sigma_x^2 + sigma_y^2 + C2)]
      //
      mu1_mu2[j-border][i-border] = mu1[j-border][i-border] * mu2[j-border][i-border];
      mu1_sq[j-border][i-border] = mu1[j-border][i-border] * mu1[j-border][i-border];
      mu2_sq[j-border][i-border] = mu2[j-border][i-border] * mu2[j-border][i-border];
      sigma12[j-border][i-border] = (sigma12[j-border][i-border] -
                                     mu1_mu2[j-border][i-border]);
      sigma1_sq[j-border][i-border] = (sigma1_sq[j-border][i-border] -
                                       mu1_sq[j-border][i-border]);
      sigma2_sq[j-border][i-border] = (sigma2_sq[j-border][i-border] -
                                       mu2_sq[j-border][i-border]);
      numerator1 = ((2 * mu1_mu2[j-border][i-border]) + C1);
      numerator2 = ((2 * sigma12[j-border][i-border]) + C2);
      denominator1 = (mu1_sq[j-border][i-border] + mu2_sq[j-border][i-border] + C1);
      denominator2 = (sigma1_sq[j-border][i-border] + sigma2_sq[j-border][i-border] + C2);
      ssim_map = (numerator1 * numerator2) / (denominator1 * denominator2);
      result = result + (float)ssim_map;

      // Calculate the Luminance
      if(PAR_OPT == 0){
        luminance      = numerator1 / denominator1;
        mean_luminance = mean_luminance + luminance;
      }
    }
  }

  // A mean SSIM index (MSSIM) is used to evaluate the overall image quality
  //
  // MSSIM(X,Y) = (1/M)SUM(SSIM(x,y))
  //
  result = result / (((height - size_window) + 1) * ((width - size_window) + 1));

  if(PAR_OPT == 0){
    // Calculate the ratio
    ratio = ((float)result) / floating_MSSIM_array[img_ptr];
    img_ptr++;

    // Calculate the Mean Luminance
    mean_luminance = mean_luminance / (((height - size_window) + 1) *
                                       ((width - size_window) + 1));

    // Find the smallest ratio and worst mean luminance
    if(first_time){
      smallest_ratio  = ratio;
      worst_luminance = (float)mean_luminance;
      first_time      = false;
    } else {
      if(ratio < smallest_ratio)
        smallest_ratio = ratio;
      if(mean_luminance < worst_luminance)
        worst_luminance = (float)mean_luminance;
    }


  } else {
    // Calculate the sum of all RATIO
    ratio      = (the_noisy_quality/result);
    sum_ratio += ratio;

    // Save floating simulation result
    floating_MSSIM_array[img_ptr] = ((float)result);
    img_ptr++;

    // Find the worst ratio (biggest)
    if(first_time){
      worst_ratio = ratio;
      first_time  = false;
    } else {
      if(ratio > worst_ratio)
        worst_ratio = ratio;
    }
  }

  // Free the memory
  for(j = 0; j < height; j++){
    delete[] original[j];
    delete[] distorted[j];
    delete[] img1_img2[j];
    delete[] img1_sq[j];
    delete[] img2_sq[j];
    if((j > (0 + border - 1)) && (j < (height - border))){
      delete[] mu1_mu2[j-border];
      delete[] mu1[j-border];
      delete[] mu2[j-border];
```

```
        delete[] mu1_sq[j-border];
        delete[] mu2_sq[j-border];
        delete[] sigma12[j-border];
        delete[] sigma1_sq[j-border];
        delete[] sigma2_sq[j-border];
      }
    }
    delete [] img1_img2;
    delete [] img1_sq;
    delete [] img2_sq;
    delete [] mu1_mu2;
    delete [] mu1;
    delete [] mu2;
    delete [] mu1_sq;
    delete [] mu2_sq;
    delete [] sigma12;
    delete [] sigma1_sq;
    delete [] sigma2_sq;
    delete [] original;
    delete [] distorted;
    img1_img2 = NULL;
    img1_sq   = NULL;
    img2_sq   = NULL;
    mu1_mu2   = NULL;
    mu1       = NULL;
    mu2       = NULL;
    mu1_sq    = NULL;
    mu2_sq    = NULL;
    sigma12   = NULL;
    sigma1_sq = NULL;
    sigma2_sq = NULL;
    original  = NULL;
    distorted = NULL;
  }

  // WARNING
  // Encountered a negative pixel in the fixed filter result
  if(SARGE_FLAG0 == 1){
    smallest_ratio = 0.0;
    SARGE_FLAG0    = 0;
  }

  if(PAR_OPT == 0){
    puterror("IRATIO",1 - smallest_ratio);
    puterror("MAX_PT",max_threshold);
    puterror("ILUMA", 1 - worst_luminance);
  } else {
    puterror("IRATIO",(sum_ratio / img_ptr));
    puterror("MAX_PT", worst_ratio);
    puterror("ILUMA", 1);
  }
}
```

E.3 Code source pour simuler les séquences vidéo

E.3.1 Fichier Xvideo.c

```
//-----------------------------------------------------------------------------
// Title          : Modele C/C++ de l'algorithm conetant une sequence video
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : Xvideo.c
// Author         : Serge CATUDAL
// Created        : 04/01/2005
// Last modified  : 10/01/2005
//-----------------------------------------------------------------------------
// Description    :
//-----------------------------------------------------------------------------
// Copyright (c) 2005 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 04/01/2005 : created
//-----------------------------------------------------------------------------

// Parameter optimization definition
```

```
// Architecture optimization definition

// Constant definitions

// C/C++ inclusions
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>

// SynapC inclusions
#include <gFix.h>
#include <Xpublic.h>

// Global external variable declaration
extern int LAST_CHECK;

// Video function
int** compute_video(int ***img_in,int img_height,int img_width,int cur_field);

// SynapC function
void  put_image(int img_num,int **img_src,int **img_deinterlaced,
                int img_height,int img_width);

// SynapC main function
void XSmain(){
  int    cur_field;
  int    first_field;
  int    nb_field;
  int    ***img_seq;
  int    **img_src;
  int    **img_deinterlaced;
  int    img_height,img_width;
  int    img_num = 0;
  int    row, col;
  int    the_int;
  float  the_float;
  float  the_height,the_width,the_nb_field,the_first_field;

  while((getinput(&the_height) == 1) && (getinput(&the_width) == 1) &&
        (getinput(&the_nb_field) == 1) && (getinput(&the_first_field) == 1)){
    // Get image size
    img_height = (int)the_height;
    img_width  = (int)the_width;

    // Get sequence number of field(s) and first field
    nb_field    = (int)the_nb_field;
    first_field = (int)the_first_field;

    // Initialize and get the original frame
    img_src = new int*[img_height];
    for(row=0; row<img_height; row++){
      img_src[row] = new int[img_width];
      for(col=0; col<img_width; col++){
        if(getinput(&the_float) == 1)
          img_src[row][col] = (int)the_float;
        else
          fprintf(stderr,"Original frame is wrong size.\n");
      }
    }

    // Initialize and get the image sequence
    img_height = img_height / 2;
    img_seq    = new int**[nb_field];
    for(cur_field=0; cur_field<nb_field; cur_field++){
      img_seq[cur_field] = new int*[img_height];
      for(row=0; row<img_height; row++){
        img_seq[cur_field][row] = new int[img_width];
        for(col=0; col<img_width; col++){
          if(getinput(&the_float) == 1)
            img_seq[cur_field][row][col] = (int)the_float;
          else
            fprintf(stderr,"Image sequence is wrong size.\n");
        }
      }
    }

    // Execute the image deinterlacer
    img_deinterlaced = compute_motion_deinterlacer(img_seq,img_height,img_width,first_field);

    // Save the result
    img_height = img_height * 2;
    put_image(img_num,img_src,img_deinterlaced,img_height,img_width);
    img_num++;
```

```
    // Free the memory
    for(row=0; row<img_height; row++){
      delete[] img_src[row];
      delete[] img_deinterlaced[row];
    }
    img_height = img_height / 2;
    for(cur_field=0; cur_field<nb_field; cur_field++){
      for(row=0; row<img_height; row++)
        delete[] img_seq[cur_field][row];
      delete[] img_seq[cur_field];
    }
    delete[] img_src;
    delete[] img_deinterlaced;
    delete[] img_seq;
    img_src           = NULL;
    img_deinterlaced  = NULL;
    img_seq           = NULL;
  }
}

//-----------------------------------------------------------------------------
// Video Algorithm Functions
//-----------------------------------------------------------------------------
// Compute the Video Algorithm
int** compute_video(int ***img_in, int img_height, int img_width, int cur_field){

  // Video Algorithm code here

  return img_out;
}

//-----------------------------------------------------------------------------
// SynapC Function
//-----------------------------------------------------------------------------
// Save the data for the Xerror algorithm
void put_image(int img_num,int **img_src,int **img_deinterlaced,
               int img_height,int img_width){
  int  row,col;
  int  itr;
  char *ctmp          = new char[10];
  char *filename      = new char[30];
  char *header_tmp    = new char[20];

  ofstream WriteBIN;

  // Save the image size
  putoutput((float)img_height);
  putoutput((float)img_width);

  if(LAST_CHECK == 1){
    // Create raw PGM image file header
    strcat(header_tmp,"P5\n");
    sprintf(ctmp,"%d",img_width);
    strcat(header_tmp,ctmp);
    strcat(header_tmp," ");
    sprintf(ctmp,"%d",img_height);
    strcat(header_tmp,ctmp);
    strcat(header_tmp,"\n255\n");

    // Create filename
    strcat(filename,"img_deint_fixed_");
    sprintf(ctmp,"%d",img_num);
    strcat(filename,ctmp);
    strcat(filename,".pgm");

    // Write raw PGM image file
    WriteBIN.open(filename, ios::binary);
    WriteBIN.seekp(0,ios::beg);
    // Write header
    for(itr=0; itr<strlen(header_tmp); itr++)
      WriteBIN.write((char *) &header_tmp[itr],sizeof(char));
  }

  // Save the original and the filtered image (interlaced pixel)
  for(row=0; row<img_height; row++){
    for(col=0; col<img_width; col++){
      putoutput((float)img_src[row][col]);
      putoutput((float)img_deinterlaced[row][col]);
      // Write each pixels into the raw PGM file
      if(LAST_CHECK == 1)
        WriteBIN.write((char *) &(unsigned char)img_deinterlaced[row][col],
                       sizeof(unsigned char));
    }
  }
  if(LAST_CHECK == 1)
    WriteBIN.close();
```

```
  // Free the memory
  delete[] ctmp;
  delete[] filename;
  delete[] header_tmp;
  ctmp       = NULL;
  filename   = NULL;
  header_tmp = NULL;
}
```

E.3.2 Fichier Xvideo_error.c

```
//-------------------------------------------------------------------------------
// Title          : Structural SIMilarity Index (SSIM Index)
// Project        : Video Processing Validation Envirronment (VPVE)
//-------------------------------------------------------------------------------
// File           : Xvideo_error.c
// Author         : Serge Catudal
// Created        : 04/01/2005
// Last modified  : 04/01/2005
//-------------------------------------------------------------------------------
// Description    : Objective Image Quality Metric
//-------------------------------------------------------------------------------
// Copyright (c) 2005 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-------------------------------------------------------------------------------
// Modification history :
// 04/01/2005 : created
//-------------------------------------------------------------------------------

// C/C++ inclusions
#include <stdio.h>
#include <stdlib.h>

// SynapC inclusions
#include <Xpublic.h>

// Size of image database
#define size_img_db  5

// Size of the circular symmetric gaussian weighting function
#define size_window 11

// Constants in the SSIM index formula
#define K1 0.01
#define K2 0.03
#define L   255

// Global internal variable declaration
float floating_MSSIM_array[size_img_db];

// Global external variable declaration
extern int PAR_OPT;       // 1 = Parameter estimation, 0 = Param estimation + ops optimization
extern int SARGE_FLAG0;   // Negative pixel detector (1 = negative pixel)

// Circular symmetric gaussian weighting function
const double window[size_window][size_window] = {
  {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000},
  {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
  {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
  {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
  {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
  {0.0003,0.0020,0.0096,0.0291,0.0567,0.0708,0.0567,0.0291,0.0096,0.0020,0.0003},
  {0.0002,0.0016,0.0077,0.0233,0.0454,0.0567,0.0454,0.0233,0.0077,0.0016,0.0002},
  {0.0001,0.0008,0.0039,0.0120,0.0233,0.0291,0.0233,0.0120,0.0039,0.0008,0.0001},
  {0.0000,0.0003,0.0013,0.0039,0.0077,0.0096,0.0077,0.0039,0.0013,0.0003,0.0000},
  {0.0000,0.0001,0.0003,0.0008,0.0016,0.0020,0.0016,0.0008,0.0003,0.0001,0.0000},
  {0.0000,0.0000,0.0000,0.0001,0.0002,0.0003,0.0002,0.0001,0.0000,0.0000,0.0000}
};

// SynapC error function
void Xerror(){
  int     border;
  double C1,C2;
  double denominator1,denominator2;
  double luminance, mean_luminance;
  double numerator1,numerator2;
  bool    first_time = true;
  int     i,j;
  int     m,n;
  int     current_threshold,max_threshold;
  int     **img1_img2,**img1_sq,**img2_sq;
```

```
double **mu1_mu2,**mu1,**mu2;
double **mu1_sq,**mu2_sq;
double **sigma12,**sigma1_sq,**sigma2_sq;
double result = 0.0;
double ssim_map;
int    **original,**distorted;
int    height,width;
float  filtered_floating;
float  floating,fixed,quality,tmp;
float  the_height,the_width;
float  the_noisy_quality;
float  ratio, smallest_ratio, sum_quality, worst_quality;
int    img_ptr;

// Variable initialization
border          = ((size_window - 1) / 2);
img_ptr         = 0;
mean_luminance  = 0.0;
smallest_ratio  = 1.0;
sum_quality     = 0.0;
worst_quality   = 1.0;

// Constants initialiazation
C1 = (K1*L)*(K1*L);
C2 = (K2*L)*(K2*L);

// MAX threshold initialization
max_threshold = 0;

while((getoutput(&floating,&the_height) == 1) && (getoutput(&floating,&the_width) == 1)){
  // WARNING
  // Encountered a negative pixel in the fixed filter result
  if(SARGE_FLAG0 == 1)
    break;

  // first_time variable re-initialization
  if(PAR_OPT == 0 && !first_time)
    first_time = true;

  // Get image characteristics
  height = (int)the_height;
  width  = (int)the_width;

  // Initialize result
  result = 0.0;

  // Calculate the square for the original and distorted image
  original  = new int*[height];
  distorted = new int*[height];
  img1_img2 = new int*[height];
  img1_sq   = new int*[height];
  img2_sq   = new int*[height];
  for(j = 0; j < height; j++){
    original[j]  = new int[width];
    distorted[j] = new int[width];
    img1_img2[j] = new int[width];
    img1_sq[j]   = new int[width];
    img2_sq[j]   = new int[width];
    for(i = 0; i < width; i++){
      if(getoutput(&tmp,&floating)!=1 || getoutput(&filtered_floating,&fixed)!=1){
        fprintf(stderr,"\n\t**_Error_**\n\n\t\tThe_size_of_the_image_is_incorrect\n");
        exit(1);
      }
      original[j][i]  = (int)floating;
      distorted[j][i] = (int)fixed;
      img1_img2[j][i] = original[j][i] * distorted[j][i];
      img1_sq[j][i]   = original[j][i] * original[j][i];
      img2_sq[j][i]   = distorted[j][i] * distorted[j][i];

      if(PAR_OPT == 0){
        // Calculate MAX value
        current_threshold = abs(((int)filtered_floating) - distorted[j][i]);
        if(current_threshold > max_threshold)
          max_threshold = current_threshold;
      }
    }
  }

  // Apply the 2D Filter on the mean intensity and the variance
  mu1       = new double*[(height - size_window) + 1];
  mu2       = new double*[(height - size_window) + 1];
  sigma12   = new double*[(height - size_window) + 1];
  sigma1_sq = new double*[(height - size_window) + 1];
  sigma2_sq = new double*[(height - size_window) + 1];
  mu1_mu2   = new double*[(height - size_window) + 1];
  mu1_sq    = new double*[(height - size_window) + 1];
```

```
mu2_sq     = new double*[(height - size_window) + 1];
for(j = (0 + border); j < (height - border); j++){
  mu1[j-border]        = new double[(width - size_window) + 1];
  mu2[j-border]        = new double[(width - size_window) + 1];
  sigma12[j-border]    = new double[(width - size_window) + 1];
  sigma1_sq[j-border]  = new double[(width - size_window) + 1];
  sigma2_sq[j-border]  = new double[(width - size_window) + 1];
  mu1_mu2[j-border]    = new double[(width - size_window) + 1];
  mu1_sq[j-border]     = new double[(width - size_window) + 1];
  mu2_sq[j-border]     = new double[(width - size_window) + 1];
  for(i = (0 + border); i < (width - border); i++){
    mu1[j-border][i-border]        = 0;
    mu2[j-border][i-border]        = 0;
    sigma12[j-border][i-border]    = 0;
    sigma1_sq[j-border][i-border]  = 0;
    sigma2_sq[j-border][i-border]  = 0;
    // Apply the circular symmetric gaussian weighting function on each pixel
    for(n = (0 - border); n < (size_window - border); n++){
      for(m = (0 - border); m < (size_window - border); m++){
        mu1[j-border][i-border] =
          mu1[j-border][i-border] +
          ((double)(original[j+n][i+m]) * window[n+border][m+border]);
        mu2[j-border][i-border] =
          mu2[j-border][i-border] +
          ((double)(distorted[j+n][i+m]) * window[n+border][m+border]);
        sigma12[j-border][i-border] =
          sigma12[j-border][i-border] +
          ((double)(img1_img2[j+n][i+m]) * window[n+border][m+border]);
        sigma1_sq[j-border][i-border] =
          sigma1_sq[j-border][i-border] +
          ((double)(img1_sq[j+n][i+m]) * window[n+border][m+border]);
        sigma2_sq[j-border][i-border] =
          sigma2_sq[j-border][i-border] +
          ((double)(img2_sq[j+n][i+m]) * window[n+border][m+border]);
      }
    }

    // SSIM(x,y) = [l(x,y)] * [c(x,y)] * [s(x,y)]
    //
    //    where
    //
    //          l(x,y) is the luminance comparison
    //          l(x,y) = (2*mu_x*mu_y + C1) / (mu_x^2 + mu_y^2 + C1)
    //
    //          c(x,y) is the contrast comparison
    //          c(x,y) = (2*sigma_x*sigma_y + C2) / (sigma_x^2 + sigma_y^2 + C2)
    //
    //          s(x,y) is the structure comparison
    //          s(x,y) = (sigma_xy + C3) / (sigma_x*sigma_y + C3)
    //
    // SSIM(x,y) = [(2*mu_x*mu_y + C1) * (2*sigma_x*sigma_y + C2)] /
    //             [(mu_x^2 + mu_y^2 + C1) * (sigma_x^2 + sigma_y^2 + C2)]
    //
    mu1_mu2[j-border][i-border] = mu1[j-border][i-border] * mu2[j-border][i-border];
    mu1_sq[j-border][i-border] = mu1[j-border][i-border] * mu1[j-border][i-border];
    mu2_sq[j-border][i-border] = mu2[j-border][i-border] * mu2[j-border][i-border];
    sigma12[j-border][i-border] = (sigma12[j-border][i-border] -
                                   mu1_mu2[j-border][i-border]);
    sigma1_sq[j-border][i-border] = (sigma1_sq[j-border][i-border] -
                                     mu1_sq[j-border][i-border]);
    sigma2_sq[j-border][i-border] = (sigma2_sq[j-border][i-border] -
                                     mu2_sq[j-border][i-border]);
    numerator1 = ((2 * mu1_mu2[j-border][i-border]) + C1);
    numerator2 = ((2 * sigma12[j-border][i-border]) + C2);
    denominator1 = (mu1_sq[j-border][i-border] + mu2_sq[j-border][i-border] + C1);
    denominator2 = (sigma1_sq[j-border][i-border] + sigma2_sq[j-border][i-border] + C2);
    ssim_map = (numerator1 * numerator2) / (denominator1 * denominator2);
    result = result + (float)ssim_map;

    // Calculate the Luminance
    if(PAR_OPT == 0){
      luminance      = numerator1 / denominator1;
      mean_luminance = mean_luminance + luminance;
    }
  }
}

// A mean SSIM index (MSSIM) is used to evaluate the overall image quality
//
// MSSIM(X,Y) = (1/M)SUM(SSIM(x,y))
//
result = result / (((height - size_window) + 1) * ((width - size_window) + 1));

if(PAR_OPT == 0){ // Operand + Parameter Optimization
  // Calculate the ratio
  ratio = ((float)result) / floating_MSSIM_array[img_ptr];
```

```
      img_ptr++;
      if(first_time){
        smallest_ratio = ratio;
        first_time     = false;
      } else {
        if(ratio < smallest_ratio)
          smallest_ratio = ratio;
      }

      // Calculate the Mean Luminance
      mean_luminance = mean_luminance / (((height - size_window) + 1) *
                                         ((width - size_window) + 1));
    } else {          // Parameter Optimization
      // Calculate the sum of all image quality
      sum_quality += result;

      // Save floating simulation result
      floating_MSSIM_array[img_ptr] = ((float)result);
      img_ptr++;

      // Find the worst image quality
      if(first_time){
        worst_quality = result;
        first_time    = false;
      } else {
        if(result < worst_quality)
          worst_quality = result;
      }
    }

    // Free the memory
    for(j = 0; j < height; j++){
      delete[] original[j];
      delete[] distorted[j];
      delete[] img1_img2[j];
      delete[] img1_sq[j];
      delete[] img2_sq[j];
      if((j > (0 + border - 1)) && (j < (height - border))){
        delete[] mu1_mu2[j-border];
        delete[] mu1[j-border];
        delete[] mu2[j-border];
        delete[] mu1_sq[j-border];
        delete[] mu2_sq[j-border];
        delete[] sigma12[j-border];
        delete[] sigma1_sq[j-border];
        delete[] sigma2_sq[j-border];
      }
    }
    delete []img1_img2;
    delete []img1_sq;
    delete []img2_sq;
    delete []mu1_mu2;
    delete []mu1;
    delete []mu2;
    delete []mu1_sq;
    delete []mu2_sq;
    delete []sigma12;
    delete []sigma1_sq;
    delete []sigma2_sq;
    delete []original;
    delete []distorted;
    img1_img2 = NULL;
    img1_sq   = NULL;
    img2_sq   = NULL;
    mu1_mu2   = NULL;
    mu1       = NULL;
    mu2       = NULL;
    mu1_sq    = NULL;
    mu2_sq    = NULL;
    sigma12   = NULL;
    sigma1_sq = NULL;
    sigma2_sq = NULL;
    original  = NULL;
    distorted = NULL;
  }

  // WARNING
  // Encountered a negative pixel in the fixed filter result
  if(SARGE_FLAG0 == 1){
    smallest_ratio = 0.0;
    SARGE_FLAG0    = 0;
  }

  if(PAR_OPT == 0){
    puterror("IRATIO",1 - smallest_ratio);
    puterror("MAX_PT",max_threshold);
```

```
    puterror("ILUMA", 1 - mean_luminance);
  } else {
    puterror("IRATIO",1 - (sum_quality / img_ptr));
    puterror("MAX_PT", 1 - worst_quality);
    puterror("ILUMA", 1);
  }
}
```

ANNEXE F

ALGORITHME DE DÉ-ENTRELACEMENT

F.1 Code source de l'algorithme de dé-entrelacement adapatatif au mouvement

F.1.1 Fichier motion.h

```
//-----------------------------------------------------------------------------
// Title          : Motion Adaptive De-interlacing
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : motion_adaptive.h
// Author         : Serge CATUDAL
// Created        : 30/11/2004
// Last modified  : 30/11/2004
//-----------------------------------------------------------------------------
// Description    : Motion Adaptive algorithm to deinterlace image
//-----------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 30/11/2004 : created
//-----------------------------------------------------------------------------

#ifndef _MOTION_ADAPTIVE_HEADER_
#define _MOTION_ADAPTIVE_HEADER_

// VPVE inclusions
#include <image_deinterlacer.h>

/**
 *  @ingroup deint_pack
 *  @brief This class is the Motion Adaptive De-interlacer
 *  @author Serge Catudal
 *  @version 1.0
 *  @date 30/11/2004
 */

class motion_adaptive : public image_deinterlacer {
 public:
  /// Constructor
  motion_adaptive();
  /// Copy Constructor
  motion_adaptive(const motion_adaptive&);
  /// Overload the equal (=) operator
  motion_adaptive& operator=(const motion_adaptive&);
  /// Destructor
  ~motion_adaptive();

  // Access functions
  void set_T_mov(int tm) {T_mov = tm;}
  void set_T_bak(int tb) {T_bak = tb;}
  void set_T_mr(int tmr) {T_mr  = tmr;}
  void set_T_br(int tbr) {T_br  = tbr;}
  void set_T_d(int td)   {T_d   = td;}
  void set_T_v(int tv)   {T_v   = tv;}

  // Image deinterlacer functions
  int** apply_deinterlacer(int ***img_in, vpve_field_type cur_field);
  int   apply_inter_field(int ***img_in, int ptr_j, int i, vpve_field_type cur_field);
  int   apply_intra_field(int ***img_in, int ptr_j, int i, int j,
                          int frame_height, vpve_field_type cur_field);
  void  generate_matlab_file(string parameter_filename, string matlab_filename,
                             int **img_src, int ***seq_in);
```

```
 private:
  /// MSSIM index
  ssim_index *my_ssim_index;
  /// Threshold for moving area
  int         T_mov;
  /// Threshold for boundary and background area
  int         T_bak;
  /// Threshold for moving region (AMPDF de-interlacer)
  int         T_mr;
  /// Threshold for boundary and background region (AMPDF de-interlacer)
  int         T_br;
  /// Distance Threshold (enhanced ELA)
  int         T_d;
  /// Vertical Threshold (enhanced ELA)
  int         T_v;
};

#endif
```

F.1.2 Fichier motion.cc

```
//-----------------------------------------------------------------------------
// Title          : Motion Adaptive De-interlacing
// Project        : Video Processing Validation Envirronment
//-----------------------------------------------------------------------------
// File           : motion_adaptive.cc
// Author         : Serge CATUDAL
// Created        : 30/11/2004
// Last modified  : 30/11/2004
//-----------------------------------------------------------------------------
// Description    : Motion Adaptive algorithm to deinterlace image
//-----------------------------------------------------------------------------
// Copyright (c) 2004 by Serge Catudal. This model is the confidential and
// proprietary property of Serge Catudal and the possession or use of this
// file requires a written license from Serge Catudal.
//-----------------------------------------------------------------------------
// Modification history :
// 30/11/2004 : created
//-----------------------------------------------------------------------------

#include <motion_adaptive.h>

//-----------------------------------------------------------------------------
// Default Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
motion_adaptive::motion_adaptive() : image_deinterlacer(VPVE_MOTION_ADAPTIVE_DEINTERLACER){
  my_ssim_index = new ssim_index();
  T_mov         = 31;          // 50
  T_bak         = T_mov - 24; // 30
  T_mr          = 55;          // 60
  T_br          = 85;          // 80
  T_d           = 50;          // 50
  T_v           = 102;         // 90
}

//-----------------------------------------------------------------------------
// Copy Constructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
motion_adaptive::motion_adaptive(const motion_adaptive &my_mad) : image_deinterlacer(my_mad){
  my_ssim_index = my_mad.my_ssim_index;
  T_mov         = my_mad.T_mov;
  T_bak         = my_mad.T_bak;
  T_mr          = my_mad.T_mr;
  T_br          = my_mad.T_br;
  T_d           = my_mad.T_d;
  T_v           = my_mad.T_v;
}

//-----------------------------------------------------------------------------
// = Operator Overloading
/**
 * @brief
 */
//-----------------------------------------------------------------------------
motion_adaptive& motion_adaptive::operator=(const motion_adaptive &my_mad){
  if(this != &my_mad){
    my_ssim_index = my_mad.my_ssim_index;
```

```
    T_mov           = my_mad.T_mov;
    T_bak           = my_mad.T_bak;
    T_mr            = my_mad.T_mr;
    T_br            = my_mad.T_br;
    T_d             = my_mad.T_d;
    T_v             = my_mad.T_v;
  }
  return *this;
}

//-----------------------------------------------------------------------------
// Destructor
/**
 * @brief
 */
//-----------------------------------------------------------------------------
motion_adaptive::~motion_adaptive(){
  delete my_ssim_index;
}

//-----------------------------------------------------------------------------
// Motion Adaptive De-interlacer Functions
//-----------------------------------------------------------------------------
/**
 * @brief Apply the Motion Adaptive de-interlacer
 */
int** motion_adaptive::apply_deinterlacer(int ***img_in, vpve_field_type cur_field){
  float alpha;
  float Z;
  int   BD[3];
  int   frame_height = img_height * 2;
  int   i,j;
  int   **img_out;
  int   itr;
  int   MBD_M[3];
  int   MBD_t[3];
  int   ptr_j;
  int   ptr_max;
  int   ptr_med;
  int   tmp_val;
  int   Z_ampdf,Z_mela;

  //-------------------------------------------------------------
  // | (k-2)th field | (k-1)th field | ( k )th field | (k+1)th field |
  //-------------------------------------------------------------
  // |   img_in[0]   |   img_in[1]   |   img_in[2]   |   img_in[3]   |
  //-------------------------------------------------------------

  // ptr_j initialization
  switch(cur_field){
  case VPVE_EVEN_FIELD:
    ptr_j = 0;
    break;
  case VPVE_ODD_FIELD:
    ptr_j = -1;
    break;
  }

  // img_out initialization
  img_out = new int*[frame_height];
  for(j=0; j<frame_height; j++){
    img_out[j] = new int[img_width];

    switch(cur_field){
    case VPVE_EVEN_FIELD:
      if(j%2 == 0){
        for(i=0; i<img_width; i++)
          img_out[j][i] = img_in[2][ptr_j][i];
      } else {
        // Initialisation de MBD_t
        MBD_t[0] = 0;
        MBD_t[1] = 0;
        MBD_t[2] = 0;

        for(i=0; i<img_width; i++){
          // (k-2)th   (k-1)th    (k)th    (k+1)th   field
          //-------  --------  --------  --------
          // | a_0 |  | c_1 |  | a_2 |  | c_3 |   ptr_j
          //-------  --------  --------  --------
          // | b_0 |            | b_2 |             ptr_j + 1
          //-------             -------
          //    i         i         i         i
          //
          //  EVEN      ODD       EVEN      ODD

          // Motion Detection
```

```
            // (BD_a, BD_b, BD_c)
            BD[0] = abs(img_in[0][ptr_j][i] - img_in[2][ptr_j][i]);
            if((ptr_j + 1) < img_height)
              BD[1] = abs(img_in[0][ptr_j+1][i] - img_in[2][ptr_j+1][i]);
            else
              BD[1] = 0;
            BD[2] = abs(img_in[1][ptr_j][i] - img_in[3][ptr_j][i]);

            // Maximum Brightness Difference
            // Max(BD_a, BD_b, BD_c)
            for(itr=0; itr<2; itr++){
              for(ptr_max=0; ptr_max<2; ptr_max++){
                if(BD[ptr_max + 1] < BD[ptr_max]){
                  tmp_val          = BD[ptr_max];
                  BD[ptr_max]      = BD[ptr_max + 1];
                  BD[ptr_max + 1] = tmp_val;
                }
              }
            }
            MBD_t[0] = BD[2];

            // 3-tap Median Filter (MBD_t, MBD_t-1, MBD_t-2)
            MBD_M[0] = MBD_t[0];
            MBD_M[1] = MBD_t[1];
            MBD_M[2] = MBD_t[2];
            for(itr=0; itr<2; itr++){
              for(ptr_med=0; ptr_med<2; ptr_med++){
                if(MBD_M[ptr_med + 1] < MBD_M[ptr_med]){
                  tmp_val             = MBD_M[ptr_med];
                  MBD_M[ptr_med]      = MBD_M[ptr_med + 1];
                  MBD_M[ptr_med + 1] = tmp_val;
                }
              }
            }

            // Save MBD_t
            MBD_t[2] = MBD_t[1];
            MBD_t[1] = MBD_t[0];

            // Threshold Decision
            if(MBD_M[1] > T_mov){
              // Moving Region
              alpha = 1;
            } else {
              if(MBD_M[1] > T_bak){
                // Boundary Area
                alpha = ((MBD_M[1] - T_bak) / (T_mov - T_bak));
                if(alpha > 1)
                  cout << "alpha_=_" << alpha << endl;
              } else {
                // Background Area
                alpha = 0;
              }
            }

            // Soft Switch
            Z_ampdf = apply_inter_field(img_in, ptr_j, i, cur_field);
            Z_mela  = apply_intra_field(img_in, ptr_j, i, j, frame_height, cur_field);
            Z       = ((1 - alpha) * Z_ampdf) + (alpha * Z_mela);

            img_out[j][i] = (int)Z;
          }
          // increase ptr_j
          ptr_j++;
        }
        //cout << "ok";
        break;
      case VPVE_ODD_FIELD:
        break;
      default:
        break;
      }
    }

    return img_out;
}

/**
 * @brief Apply the AMPDF deinterlacer
 */
int motion_adaptive::apply_inter_field(int ***img_in, int ptr_j,
                                       int i, vpve_field_type cur_field){
    int AD,min_AD;
    int itr;
    int median[3];
    int ptr_med;
```

```
int tmp_val;
int Z_ampdf;
int Z_cand1, pos_x_Z_cand1, pos_y_Z_cand1;
int Z_cand2, pos_x_Z_cand2, pos_y_Z_cand2;

//-------------------------------------------------------------
// | (k-1)th field | ( k )th field | (k+1)th field |
//-------------------------------------------------------------
// |    img_in[1]    |    img_in[2]    |    img_in[3]    |
//-------------------------------------------------------------

switch(cur_field){
case VPVE_EVEN_FIELD:{
  // Step 1 -> Coarse Search

  // Field (k-i)th
  //--------------------
  // |   |   | 1 |   |   |  ptr_j-1
  //--------------------
  // | 2 |   | 3 |   | 4 |  ptr_j
  //--------------------
  // |   |   | 5 |   |   |  ptr_j+1
  //--------------------
  //  i-2      i      i+2

  // AD (Absolute Difference) for 5 potential pixels
  if((ptr_j-1) < 0){
    min_AD   = 255;
  } else {
    Z_cand1        = img_in[1][ptr_j-1][i];
    min_AD         = abs(img_in[1][ptr_j-1][i] - img_in[3][ptr_j][i]);
    pos_x_Z_cand1 = i;
    pos_y_Z_cand1 = ptr_j-1;
  }

  if((i-2) < 0)
    AD = 255;
  else
    AD = abs(img_in[1][ptr_j][i-2] - img_in[3][ptr_j][i]);
  if(AD < min_AD){
    Z_cand1        = img_in[1][ptr_j][i-2];
    min_AD         = AD;
    pos_x_Z_cand1 = i-2;
    pos_y_Z_cand1 = ptr_j;
  }

  AD = abs(img_in[1][ptr_j][i] - img_in[3][ptr_j][i]);
  if(AD < min_AD){
    Z_cand1        = img_in[1][ptr_j][i];
    min_AD         = AD;
    pos_x_Z_cand1 = i;
    pos_y_Z_cand1 = ptr_j;
  }

  if((i+2) > (img_width - 1))
    AD = 255;
  else
    AD = abs(img_in[1][ptr_j][i+2] - img_in[3][ptr_j][i]);
  if(AD < min_AD){
    Z_cand1        = img_in[1][ptr_j][i+2];
    min_AD         = AD;
    pos_x_Z_cand1 = i+2;
    pos_y_Z_cand1 = ptr_j;
  }

  if((ptr_j+1) > (img_height - 1))
    AD = 255;
  else
    AD = abs(img_in[1][ptr_j+1][i] - img_in[3][ptr_j][i]);
  if(AD < min_AD){
    Z_cand1        = img_in[1][ptr_j+1][i];
    min_AD         = AD;
    pos_x_Z_cand1 = i;
    pos_y_Z_cand1 = ptr_j+1;
  }

  // | Z_ref - Z_cand1 | < T_mr
  if(min_AD < T_mr){
    // 3-tap Median Filter (Z_ref, Z_cand1, Z_k-1)
    median[0] = img_in[3][ptr_j][i];
    median[1] = Z_cand1;
    median[2] = img_in[1][ptr_j][i];
    for(itr=0; itr<2; itr++){
      for(ptr_med=0; ptr_med<2; ptr_med++){
        if(median[ptr_med + 1] < median[ptr_med]){
          tmp_val              = median[ptr_med];
```

```
        median[ptr_med]     = median[ptr_med + 1];
        median[ptr_med + 1] = tmp_val;
      }
    }
  }
  Z_ampdf = median[1];
} else {
  // Step 2 -> Fine Search 1

  // Field (k)th
  //-------------
  // | 1 |   | 3 | y
  //-------------     (Z_cand1)
  // | 2 |   | 4 | y+1
  //-------------
  //  x-1     x+1
  //   (Z_cand1)

  // AD (Absolute Difference) for 4 potential pixels
  if((pos_x_Z_cand1-1) < 0){
    min_AD = 255;
  } else {
    Z_cand2       = img_in[2][pos_y_Z_cand1][pos_x_Z_cand1-1];
    min_AD        = abs(img_in[2][pos_y_Z_cand1][pos_x_Z_cand1-1] - img_in[3][ptr_j][i]);
    pos_x_Z_cand2 = pos_x_Z_cand1-1;
    pos_y_Z_cand2 = pos_y_Z_cand1;
    if((pos_y_Z_cand1+1) < img_height){
      AD      = abs(img_in[2][pos_y_Z_cand1+1][pos_x_Z_cand1-1] - img_in[3][ptr_j][i]);
      if(AD < min_AD){
        Z_cand2       = img_in[2][pos_y_Z_cand1+1][pos_x_Z_cand1-1];
        min_AD        = AD;
        pos_y_Z_cand2 = pos_y_Z_cand1+1;
      }
    }
  }

  if((pos_x_Z_cand1+1) < img_width){
    AD = abs(img_in[2][pos_y_Z_cand1][pos_x_Z_cand1+1] - img_in[3][ptr_j][i]);
    if(AD < min_AD){
      Z_cand2       = img_in[2][pos_y_Z_cand1][pos_x_Z_cand1+1];
      min_AD        = AD;
      pos_x_Z_cand2 = pos_x_Z_cand1+1;
      pos_y_Z_cand2 = pos_y_Z_cand1;
    }
    if((pos_y_Z_cand1+1) < img_height){
      AD = abs(img_in[2][pos_y_Z_cand1+1][pos_x_Z_cand1+1] - img_in[3][ptr_j][i]);
      if(AD < Z_cand2){
        Z_cand2       = img_in[2][pos_y_Z_cand1+1][pos_x_Z_cand1+1];
        min_AD        = AD;
        pos_x_Z_cand2 = pos_x_Z_cand1+1;
        pos_y_Z_cand2 = pos_y_Z_cand1+1;
      }
    }
  }

  // | Z_ref - Z_cand1 | < T_mr
  if(min_AD < T_br){
    Z_ampdf = Z_cand2;
  } else {
    // Step 3 -> Fine Search 2
    //cerr << "Step 3 -> Fine Search 2" << endl;

    // Field (k)th                    Field (k-1)th
    //-------------                   -------------
    // |   |   |   | y-1               |   | 3 |   | y-1
    //-------------     (Z_cand2)     -------------     (Z_cand2)
    // | 1 |   | 2 | y                 |   | 4 |   | y
    //-------------                   -------------
    //  x-1     x+1                         x
    //   (Z_cand2)                    (Z_cand2)

    if((pos_x_Z_cand2-1) < 0){
      min_AD = 255;
    } else {
      min_AD  = abs(img_in[2][pos_y_Z_cand2][pos_x_Z_cand2-1] - img_in[3][ptr_j][i]);
      Z_ampdf = img_in[2][pos_y_Z_cand2][pos_x_Z_cand2-1];
    }

    if((pos_x_Z_cand2+1) < img_width){
      AD = abs(img_in[2][pos_y_Z_cand2][pos_x_Z_cand2+1] - img_in[3][ptr_j][i]);
      if(AD < min_AD){
        min_AD  = AD;
        Z_ampdf = img_in[2][pos_y_Z_cand2][pos_x_Z_cand2+1];
      }
    }
```

```
        if((pos_y_Z_cand2-1) >= 0){
          AD = abs(img_in[1][pos_y_Z_cand2-1][pos_x_Z_cand2] - img_in[3][ptr_j][i]);
          if(AD < min_AD){
            min_AD   = AD;
            Z_ampdf = img_in[1][pos_y_Z_cand2-1][pos_x_Z_cand2];
          }
        }

        AD = abs(img_in[1][pos_y_Z_cand2][pos_x_Z_cand2] - img_in[3][ptr_j][i]);
        if(AD < min_AD)
          Z_ampdf = img_in[1][pos_y_Z_cand2][pos_x_Z_cand2];
      }
    }
  }
  break;
  case VPVE_ODD_FIELD:
    Z_ampdf = 0;
    break;
  default:
    break;
  }

  return Z_ampdf;
}

/**
 * @brief Apply the enhanced ELA deinterlacer
 */
int motion_adaptive::apply_intra_field(int ***img_in, int ptr_j, int i,
                                       int j, int frame_height, vpve_field_type cur_field){
  bool  dominant = false;
  float interpolated_val;
  int   D_d1, D_d2;
  int   dir_dif[5];
  int   itr;
  int   line_buf_up[5];
  int   line_buf_down[5];
  int   median[3];
  int   n;
  int   ptr_up, ptr_down;
  int   ptr_med;
  int   smallest, smallest_1, smallest_2;
  int   tmp_val;
  int   Z_mela;

  //------------------
  // | (k)th field |
  //------------------
  // |   img_in[2]   |
  //------------------

  switch(cur_field){
  case VPVE_EVEN_FIELD:{
    // Two neighboring scan lines
    // Generate the two scan lines
    if(j == frame_height - 1){
      // Buttom scan line
      for(n=-2; n<=2; n++)
        line_buf_down[n+2] = 0;
      // Top scan line
      if(i == 0){
        line_buf_up[0] = 0;
        line_buf_up[1] = 0;
        for(n=0; n<=2; n++)
          line_buf_up[n+2] = img_in[2][ptr_j][i+n];
      } else if(i == 1){
        line_buf_up[0] = 0;
        for(n=-1; n<=2; n++)
          line_buf_up[n+2] = img_in[2][ptr_j][i+n];
      } else if(i == img_width - 2){
        for(n=-2; n<=1; n++)
          line_buf_up[n+2] = img_in[2][ptr_j][i+n];
        line_buf_up[4] = 0;
      } else if(i == img_width - 1){
        for(n=-2; n<=0; n++)
          line_buf_up[n+2] = img_in[2][ptr_j][i+n];
        line_buf_up[3] = 0;
        line_buf_up[4] = 0;
      } else {
        for(n=-2; n<=2; n++)
          line_buf_up[n+2] = img_in[2][ptr_j][i+n];
      }
    } else {
      if(i == 0){
        line_buf_up[0]   = 0;
        line_buf_down[0] = 0;
```

```
    line_buf_up[1]   = 0;
    line_buf_down[1] = 0;
    for(n=0; n<=2; n++){
      line_buf_up[n+2]   = img_in[2][ptr_j][i+n];
      line_buf_down[n+2] = img_in[2][ptr_j+1][i+n];
    }
  } else if(i == 1){
    line_buf_up[0]   = 0;
    line_buf_down[0] = 0;
    for(n=-1; n<=2; n++){
      line_buf_up[n+2]   = img_in[2][ptr_j][i+n];
      line_buf_down[n+2] = img_in[2][ptr_j+1][i+n];
    }
  } else if(i == img_width - 2){
    for(n=-2; n<=1; n++){
      line_buf_up[n+2]   = img_in[2][ptr_j][i+n];
      line_buf_down[n+2] = img_in[2][ptr_j+1][i+n];
    }
    line_buf_up[4]   = 0;
    line_buf_down[4] = 0;
  } else if(i == img_width - 1){
    for(n=-2; n<=0; n++){
      line_buf_up[n+2]   = img_in[2][ptr_j][i+n];
      line_buf_down[n+2] = img_in[2][ptr_j+1][i+n];
    }
    line_buf_up[3]   = 0;
    line_buf_down[3] = 0;
    line_buf_up[4]   = 0;
    line_buf_down[4] = 0;
  } else {
    for(n=-2; n<=2; n++){
      line_buf_up[n+2]   = img_in[2][ptr_j][i+n];
      line_buf_down[n+2] = img_in[2][ptr_j+1][i+n];
    }
  }
}

// Directional corolation calculation (a,b,c,d,e)
// 5 directional differences
//    a          b          c          d          e
// dir_dif[0] dir_dif[1] dir_dif[2] dir_dif[3] dir_dif[4]
dir_dif[0] = abs(line_buf_up[0] - line_buf_down[4]);
dir_dif[1] = abs(line_buf_up[1] - line_buf_down[3]);
dir_dif[2] = abs(line_buf_up[2] - line_buf_down[2]);
dir_dif[3] = abs(line_buf_up[3] - line_buf_down[1]);
dir_dif[4] = abs(line_buf_up[4] - line_buf_down[0]);

if(dir_dif[2] < T_v){ // Smooth region
  ptr_up   = 2;
  ptr_down = 2;
  dominant = false;
} else {              // Edge region
  // Min(a, b, c, d, e)
  // Smallest difference step 1
  if(dir_dif[0] < dir_dif[1])
    smallest_1 = 0;
  else
    smallest_1 = 1;
  if(dir_dif[3] < dir_dif[4])
    smallest_2 = 3;
  else
    smallest_2 = 4;
  // Smallest difference step 2
  if(dir_dif[smallest_1] < dir_dif[smallest_2]){
    // Smallest difference step 3
    if(dir_dif[smallest_1] < dir_dif[2])
      smallest = smallest_1;
    else
      smallest = 2;
  } else {
    // Smallest difference step 3
    if(dir_dif[smallest_2] < dir_dif[2])
      smallest = smallest_2;
    else
      smallest = 2;
  }

  // Irregular?
  switch(smallest){
  case 0: // min = a
    D_d1 = abs(dir_dif[0] - dir_dif[3]);
    D_d2 = abs(dir_dif[0] - dir_dif[4]);
    ptr_up   = 0;
    ptr_down = 4;
    if(D_d1 > T_d && D_d2 > T_d){ // Dominant directional edge
      dominant = true;
```

```
        } else {                          // Non-dominant directional edge
          dominant = false;
        }
        break;
      case 1: // min = b
        D_d1 = abs(dir_dif[1] - dir_dif[3]);
        D_d2 = abs(dir_dif[1] - dir_dif[4]);
        ptr_up   = 1;
        ptr_down = 3;
        if(D_d1 > T_d && D_d2 > T_d){ // Dominant directional edge
          dominant = true;
        } else {                          // Non-dominant directional edge
          dominant = false;
        }
        break;
      case 2: // min = c
        ptr_up   = 2;
        ptr_down = 2;
        dominant = false;
        break;
      case 3: // min = d
        D_d1 = abs(dir_dif[3] - dir_dif[0]);
        D_d2 = abs(dir_dif[3] - dir_dif[1]);
        ptr_up   = 3;
        ptr_down = 1;
        if(D_d1 > T_d && D_d2 > T_d){ // Dominant directional edge
          dominant = true;
        } else {                          // Non-dominant directional edge
          dominant = false;
        }
        break;
      case 4: // min = e
        D_d1 = abs(dir_dif[4] - dir_dif[0]);
        D_d2 = abs(dir_dif[4] - dir_dif[1]);
        ptr_up   = 4;
        ptr_down = 0;
        if(D_d1 > T_d && D_d2 > T_d){ // Dominant directional edge
              dominant = true;
        } else {                          // Non-dominant directional edge
          dominant = false;
        }
        break;
      default:
        break;
      }
    }

    // Interpolated Value
    interpolated_val = ((float)(line_buf_up[ptr_up] + line_buf_down[ptr_down])) / 2;

    if(dominant){
      // Median filter
      // y = Median[x(k-1,n),x(k+1,n),ELA]
      median[0] = line_buf_up[2];
      median[1] = line_buf_down[2];
      median[2] = (int)interpolated_val;
      for(itr=0; itr<2; itr++){
        for(ptr_med=0; ptr_med<2; ptr_med++){
          if(median[ptr_med + 1] < median[ptr_med]){
            tmp_val              = median[ptr_med];
            median[ptr_med]      = median[ptr_med + 1];
            median[ptr_med + 1] = tmp_val;
          }
        }
      }
      Z_mela = median[1];
    } else {
      Z_mela = (int)interpolated_val;
    }
  }
  break;
  case VPVE_ODD_FIELD:
    Z_mela = 0;
    break;
  default:
    break;
  }

  return Z_mela;
}

/**
 * @brief Generate the Motion Adaptive de-interlacer matlab files
 */
void motion_adaptive::generate_matlab_file(string parameter_filename,string matlab_filename,
                                           int **img_src,int ***seq_in){
```

```
char            *filename;
float           img_quality = 0;
int             mov_t, mov_start, mov_end;
int             bak_t, bak_start, bak_end;
int             j;
int             **img_deinterlaced;
int             nb_fields;
string          first_field_type;
string          tampon;
string          tmp_filename;
vpve_field_type cur_field;

ifstream ReadTXT;
ofstream WriteMatlab_results;

// Parameter file content:
//
// Image Filename
// Nb field per sequence
// First field type (even or odd)
// Moving Region Threshold start
// Moving Region Threshold end
// Boundary and Background Region Threshold start
// Boundary and Background Region Threshold end
//
// Example of the content of the parameter file:
//
// football_09.pgm              <- Image filename
// 4                            <- Nb fields per sequence
// even                         <- First field type
// 0                            <- Moving Area Threshold start
// 120                          <- Moving Area Threshold end
// 0                            <- Boundary and Background Area Threshold start
// 100                          <- Boundary and Background Area Threshold end

cout << "Reading Parameters:" << endl;

// Open parameter file
ReadTXT.open(parameter_filename.c_str());

// Save matlab results file
filename = new char[40];
strcat(filename,matlab_filename.c_str());
strcat(filename,"_results.m");
WriteMatlab_results.open(filename);
delete[] filename;
filename = NULL;

if(!ReadTXT.fail() && !WriteMatlab_results.fail()){
  // Load the source image
  ReadTXT >> tmp_filename >> tampon;

  // Load image sequence informations
  ReadTXT >> nb_fields >> first_field_type;
  cout << "\tImage_sequence_informations:" << endl;
  cout << "\t\tNb_fields_per_sequence:_" << nb_fields << endl;
  cout << "\t\tFirst_field_type:_______" << first_field_type << endl;
  if(first_field_type == "even")
    cur_field = VPVE_EVEN_FIELD;
  else
    cur_field = VPVE_ODD_FIELD;

  // Load Distance threshold parameters
  ReadTXT >> mov_start >> mov_end;
  cout << "\tMoving_Area_threshold_parameters:_" << endl;
  cout << "\t\tStart:_" << mov_start << endl;
  cout << "\t\tEnd___:_" << mov_end << endl;

  // Load Vertical threshold parameters
  ReadTXT >> bak_start >> bak_end;
  cout << "\tBoundary_and_Background_Area_threshold_parameters:_" << endl;
  cout << "\t\tStart:_" << bak_start << endl;
  cout << "\t\tEnd___:_" << bak_end << endl;
  cout <<"-------------------------------------------------------------------------\n";

  // Initialize the ssim_index object
  my_ssim_index->set_image_height(img_height);
  my_ssim_index->set_image_width(img_width);
  my_ssim_index->set_ssim_map();

  // Filter the image with different Hybrid filter parameters
  // Motion Detector
  WriteMatlab_results << "motion_md=[";
  // Intra-Field : Modified ELA
  // WriteMatlab_results << "motion_ela=[";
  // Inter-Field : AMPDF
```

```
    // WriteMatlab_results << "motion_ampdf=[";
    for(mov_t = mov_start; mov_t <= mov_end; mov_t = mov_t + 2){
      for(bak_t = bak_start; bak_t <= bak_end; bak_t = bak_t + 2){
        // Motion Detector
        set_T_mov(mov_t);
        if((mov_t - bak_t) < 2)
          set_T_bak(2);
        else
          set_T_bak((mov_t - bak_t));
        // Intra-Field : Modified ELA
        // set_T_d(mov_t);
        // set_T_v(bak_t);
        // Inter-Field : AMPDF
        // set_T_mr(mov_t);
        // set_T_br(bak_t);
        img_height        = img_height / 2;
        img_deinterlaced = apply_deinterlacer(seq_in,cur_field);
        img_quality = (float)(my_ssim_index->compute_ssim_index(img_src.img_deinterlaced));
        if(bak_t != bak_end)
          WriteMatlab_results << img_quality << " ";
        else
          WriteMatlab_results << img_quality;

        // Free the memory
        img_height = img_height * 2;
        for(j=0; j<img_height; j++)
          delete[] img_deinterlaced[j];
        delete[] img_deinterlaced;
        img_deinterlaced = NULL;
      }
      if(mov_t != mov_end)
        WriteMatlab_results << ";\n";
    }
    WriteMatlab_results << "];";
  }

  ReadTXT.close();
  WriteMatlab_results.close();
}
```